Проектирование ПО

Лекция 14: Проектирование распределённых приложений. Часть 2

Тимофей Брыксин
timofey.bryksin@gmail.com

Representational State Transfer (REST)

- Модель клиент-сервер
- Отсутствие состояния
- Кэширование
- Единообразие интерфейса
- Слои
- Код по требованию

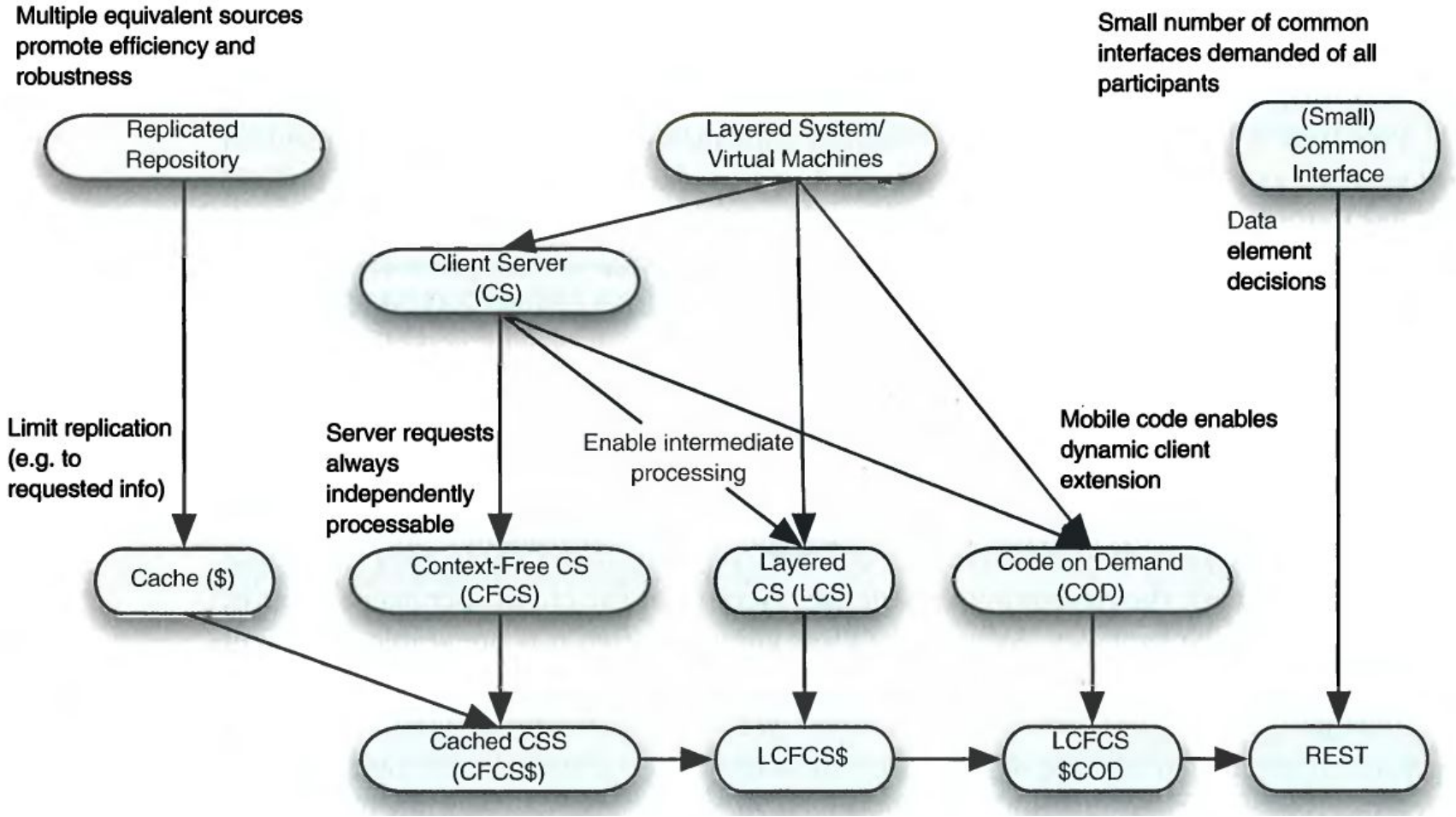
Интерфейс сервиса

- Коллекции
  - http://api.example.com/resources/
- Элементы
  - http://api.example.com/resources/item/17
- HTTP-методы
  - GET
  - PUT
  - POST
  - DELETE

Примеры

- GET /book/ — получить список всех книг
- GET /book/3/ — получить книгу номер 3
- PUT /book/3/ — заменить книгу (данные в теле запроса)
- POST /book – добавить книгу (данные в теле запроса)
- DELETE /book/3 – удалить книгу

- POST /book/ – добавить книгу (данные в теле запроса)
- POST /book/3 – изменить книгу (данные в теле запроса)
- POST /book/3 – удалить книгу (тело запроса пустое)


Multiple equivalent sources promote efficiency and robustness
Small number of common interfaces demanded of all participants
Replicated Repository
Layered System/ Virtual Machines
(Small) Common Interface
Client Server (CS)
Data element decisions
Limit replication (e.g. to requested info)
Server requests always independently processable
Enable intermediate processing
Mobile code enables dynamic client extension
Cache ($)
Context-Free CS (CFCS)
Layered CS (LCS)
Code on Demand (COD)
Cached CSS (CFCS$)
LCFCS$
LCFCS $COD
REST

Достоинства

- надёжность
- производительность
- масштабируемость
- прозрачность системы взаимодействия
- простота интерфейсов
- портативность компонентов
- лёгкость внесения изменений

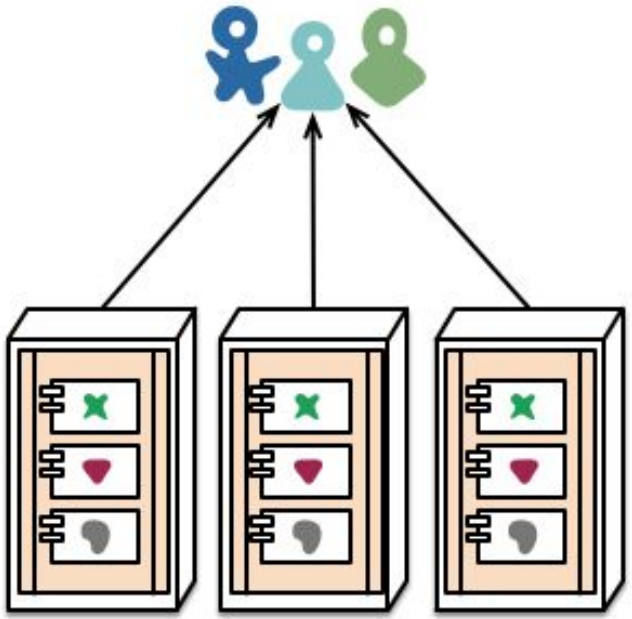
Микросервисы

- набор небольших сервисов
  - разные языки и технологии
- каждый в собственном процессе
  - независимое развёртывание
  - децентрализованное управление
- легковесные коммуникации

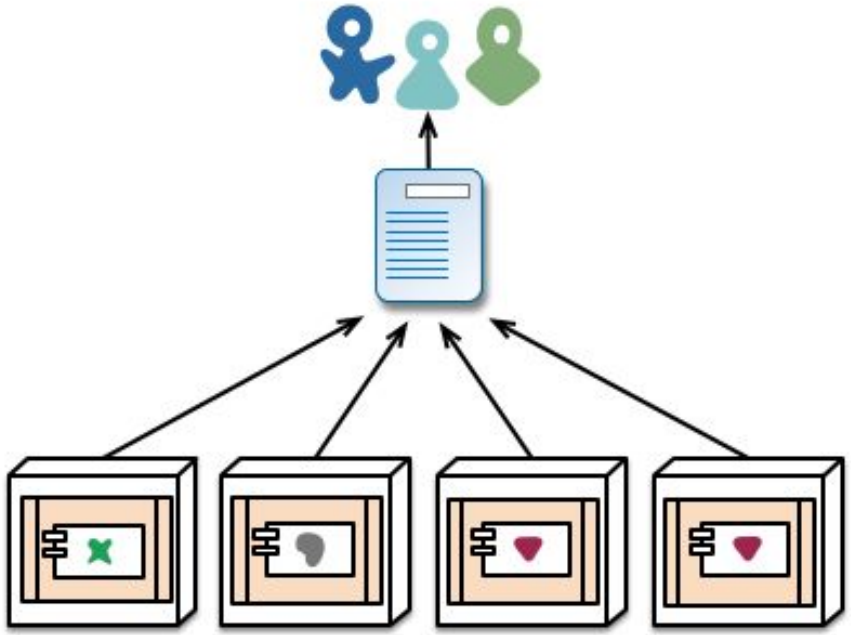
Монолитные приложения

- большой и сложный MVC
- единый процесс разработки и стек технологий
- сложная архитектура
- сложно масштабировать
- сложно вносить изменения

Разбиение на сервисы



monolith - multiple modules in the same process



microservices - modules running in different processes

Основные особенности

- Микросервисы и SOA
- Smart endpoints and dumb pipes
- Асинхронные вызовы
- Децентрализованное управление данными
- Автоматизация инфраструктуры
- Design for failure
- Эволюционный дизайн

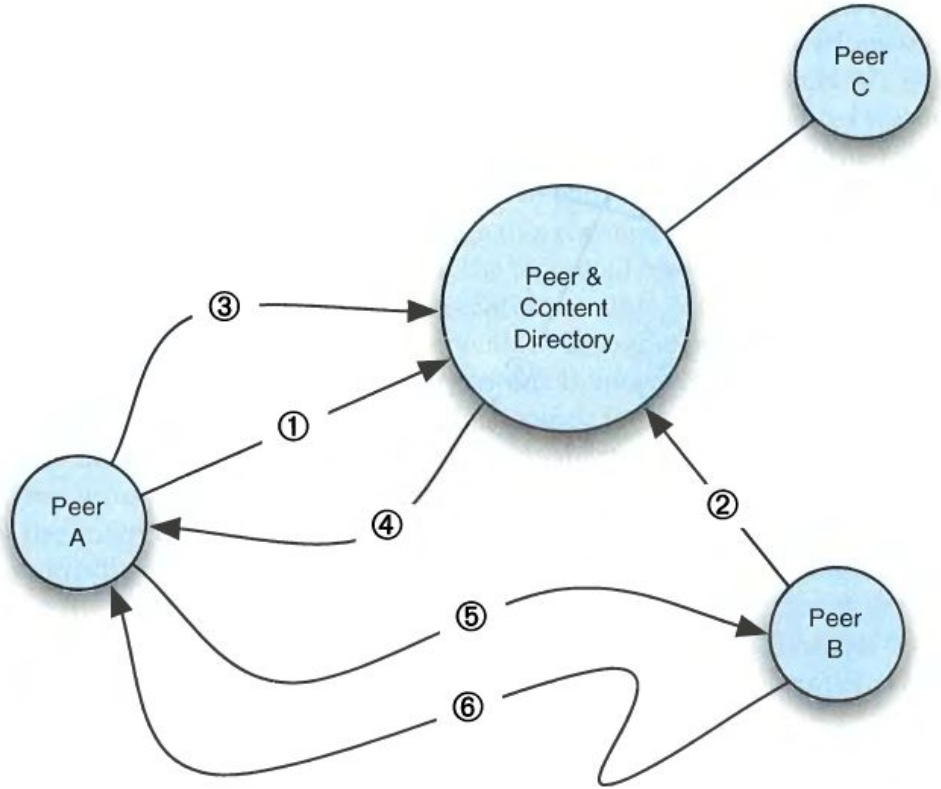
Основные проблемы

- сложности выделения границ сервисов
- перенос логики на связи между сервисами
  - большой обмен данными
  - нетривиальные зависимости
- нетривиальная инфраструктура
- нетривиальная переиспользуемость кода

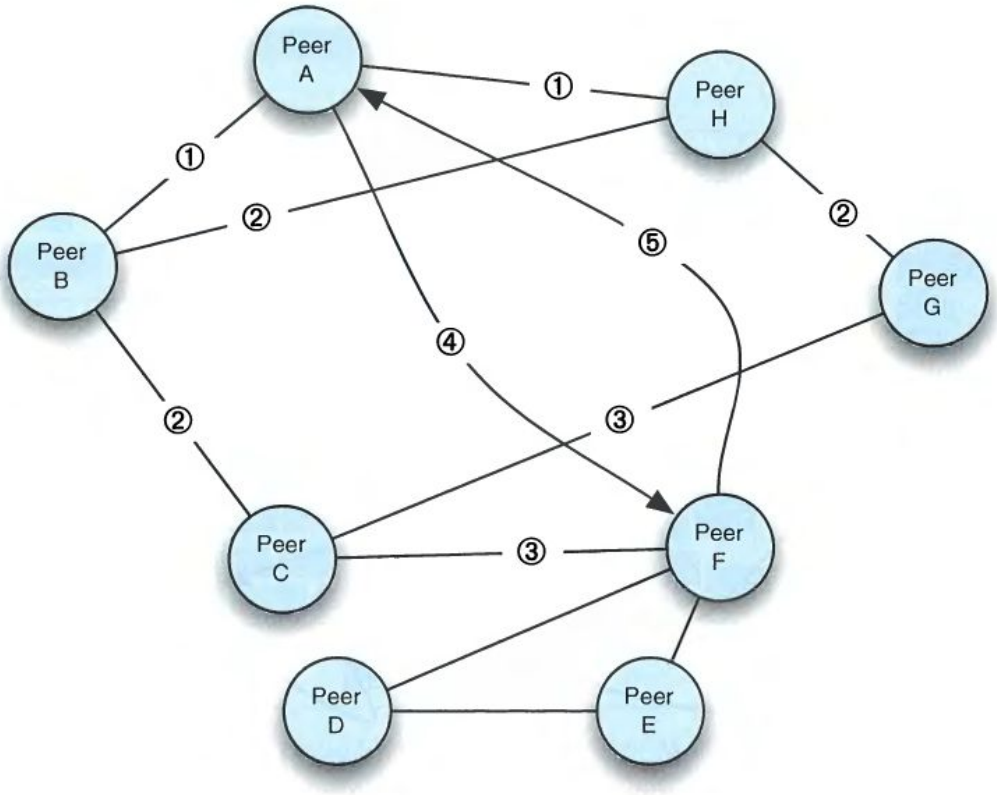
Архитектура Peer-to-Peer

- децентрализованный и самоорганизующийся сервис
- динамическая балансировка нагрузки
  - вычислительные ресурсы
  - хранилища данных
- динамическое изменение состава участников

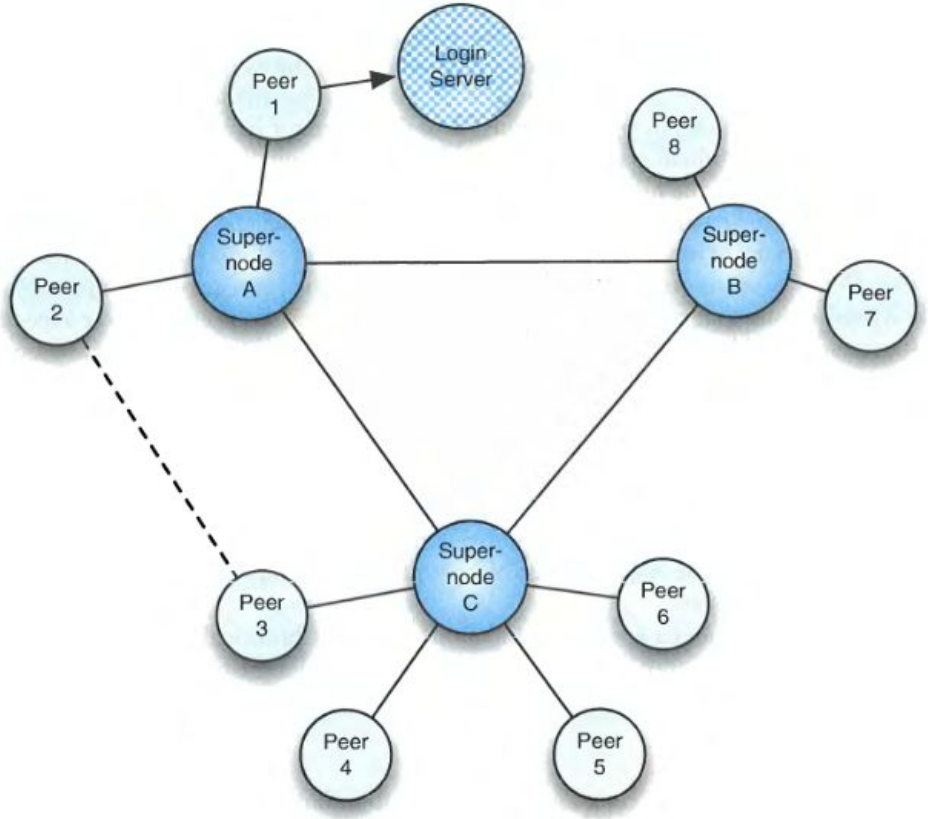
Napster: hybrid client-server/P2P



Gnutella: pure decentralized P2P



Skype: Overlayed P2P



BitTorrent : Resource Trading P2P

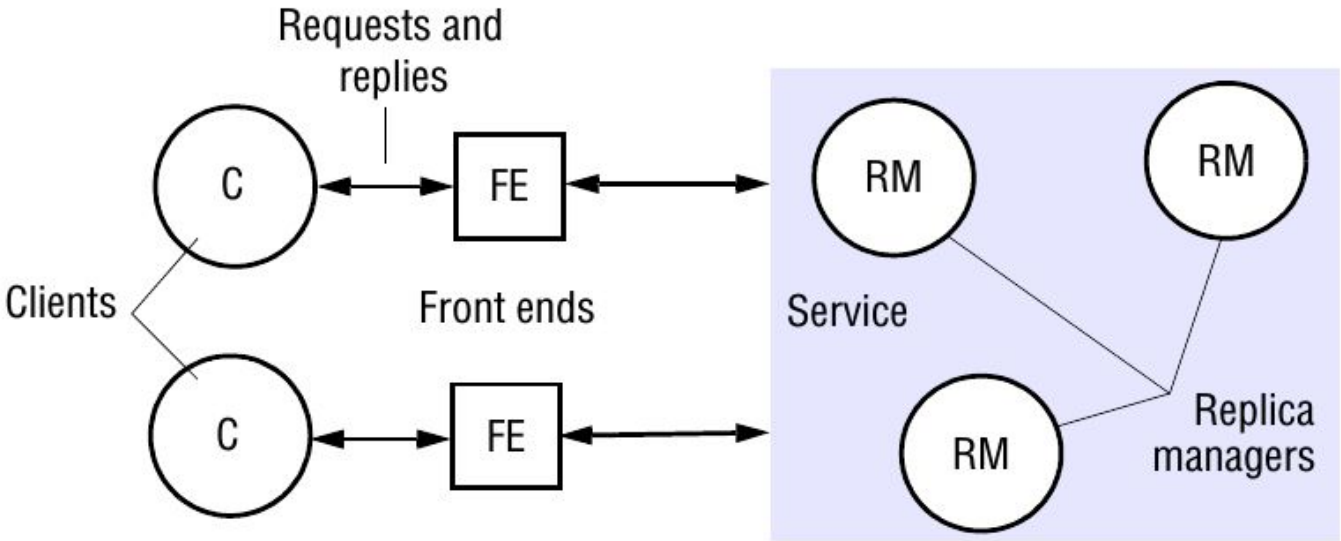
- обмен сегментами
- поиск не входит в протокол
- трекеры
- метаданные
- управление приоритетами
- бестрекерная реализация

Репликация данных

- синхронизация данных между разными процессами
  - прозрачность
- мотивация
  - повышение производительности
  - повышение доступности данных
  - отказоустойчивость

Модель системы

1. запрос
2. координирование
3. выполнение
4. соглашение
5. ответ



Отказоустойчивость

- доступность при отказе серверов
- клиенты не видят последствий репликации

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

Линеаризуемость

- последовательность действий клиента i: $\{o_{i1}, o_{i2}, o_{i3}\}$
- операции выполняются синхронно
- сервер упорядочивает операции всех клиентов
  - например, $\{o_{20}, o_{21}, o_{10}, o_{22}, o_{11}, o_{12}\}$
- линеаризуемость:
  - упорядочение как на одном наборе данных
  - порядок операций согласуется с временем их исполнения
- не должна поддерживаться транзакционность
  - может ломать целостность промежуточных состояний

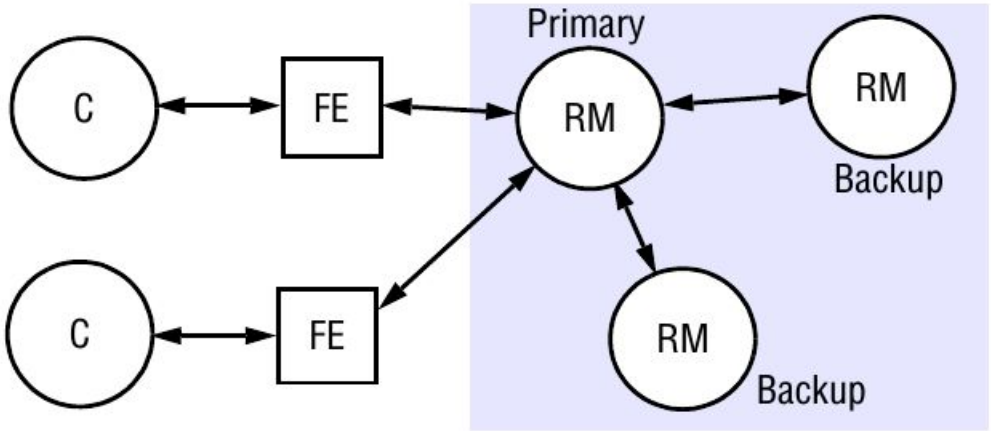
Последовательная согласованность

- более слабый критерий
  - упорядочение как на одном наборе данных
  - порядок операций согласуется с порядком исполнения на каждом клиенте

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

Пассивная репликация

- основной репликатор + бэкапы
- front-end общается только с основным
- тот выполняет операции и рассылает уведомления на бэкапы



Линеаризуемость

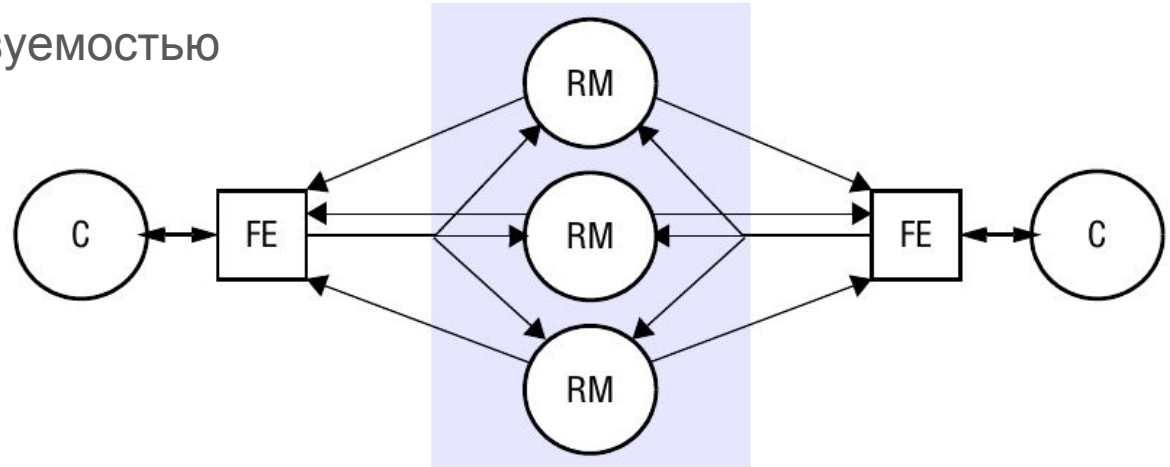
- ок, если основной репликатор работает
- если основной репликатор отказывает
  - клиенты разговаривают с одним бэкапом
  - бэкапы договариваются о конфигурации основного

Особенности пассивной репликации

- основной репликатор может вести себя недетерминированно
- система может пережить отказ n из n+1 репликатора
- front-end не должен быть особо умным
- накладные расходы на коммуникации между репликаторам
- запросы на чтение напрямую репликаторам

Активная репликация

- репликаторы как конечные автоматы, организованные в группу
- параллельное выполнение запроса всеми
- рассылка ответа
- принятие решения front-end’ом
- проблемы с линеаризуемостью



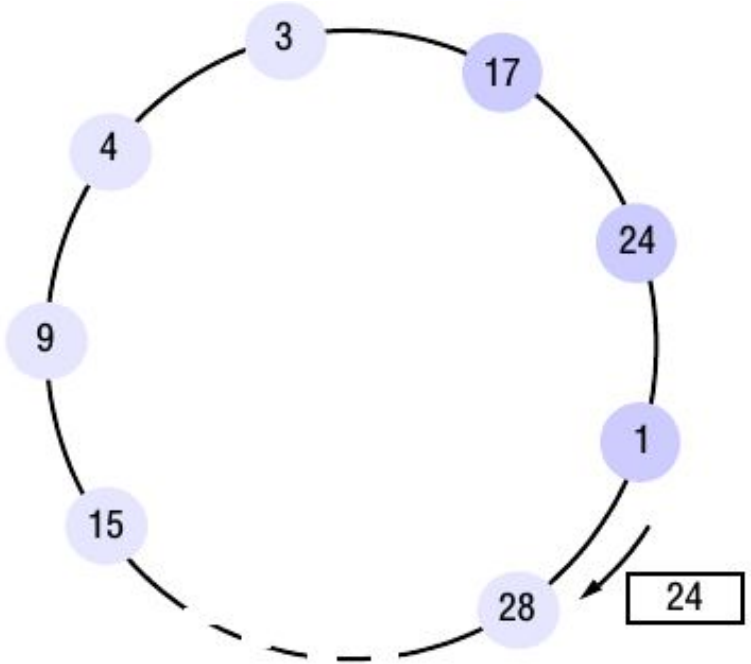
Алгоритмы голосования

- назначение определённой роли одному из участников
- все участники соглашаются с выбором
- единый результат, даже при нескольких процессах выбора

E1: (safety)	A participant process p_i has $elected_i = \perp$ or $elected_i = P$, where P is chosen as the non-crashed process at the end of the run with the largest identifier.
E2: (liveness)	All processes p_i participate and eventually either set $elected_i \neq \perp$ – or crash.

The ring-based algorithm

- узлы соединены в кольцо
- система асинхронная
- не происходит отказов
- статус узлов
  - участник голосования
  - не участник голосования
- посылаемые сообщения
  - процесс голосования
  - оповещение результатов



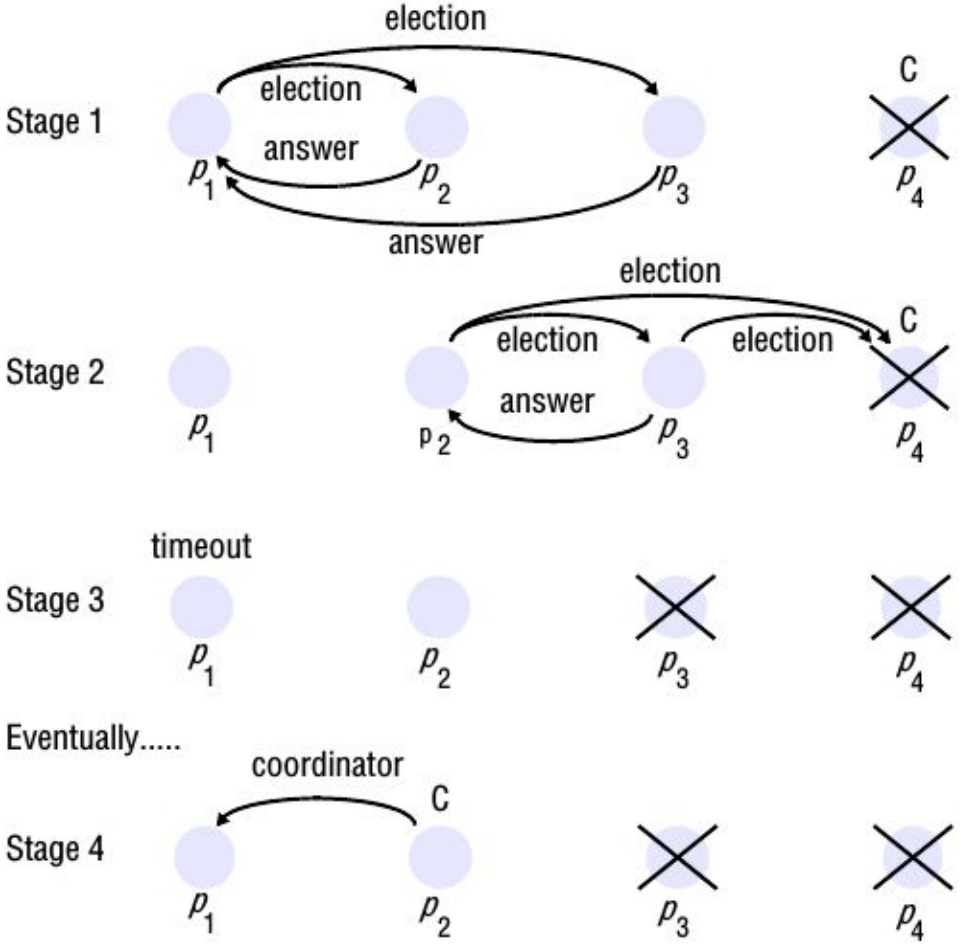
The bully algorithm

- надёжные каналы связи
- допускает отказ участников голосования
- синхронная система
  - таймауты как средство детектирования отказа
- процесс знает о более важных процессах
  - и может с ними взаимодействовать
- сообщения
  - оповещение о начале голосования
  - ответ на оповещение
  - результат голосования

Алгоритм

- процесс с наибольшим id объявляет себя
  - даже если координатор уже есть
- процесс с меньшим id запускает голосование
  - отправляет запросы всем, кто выше, и ждёт ответа
  - если не приходит за время T1, объявляет себя
  - если приходит, то ждёт время T2 сообщения о результате
    - если не приходит -- всё заново
- если процесс получает результат, то сохраняет его у себя
- если процесс получает сообщение о голосовании, посылает ответ и запускает голосование (если не запускал ранее)

Пример


election
election
Stage 1
answer
C
p_1
p_2
p_3
p_4
answer
election
election
election
C
Stage 2
answer
p_1
p_2
p_3
p_4
timeout
Stage 3
p_1
p_2
p_3
p_4
Eventually.....
coordinator
C
Stage 4
p_1
p_2
p_3
p_4

Результаты

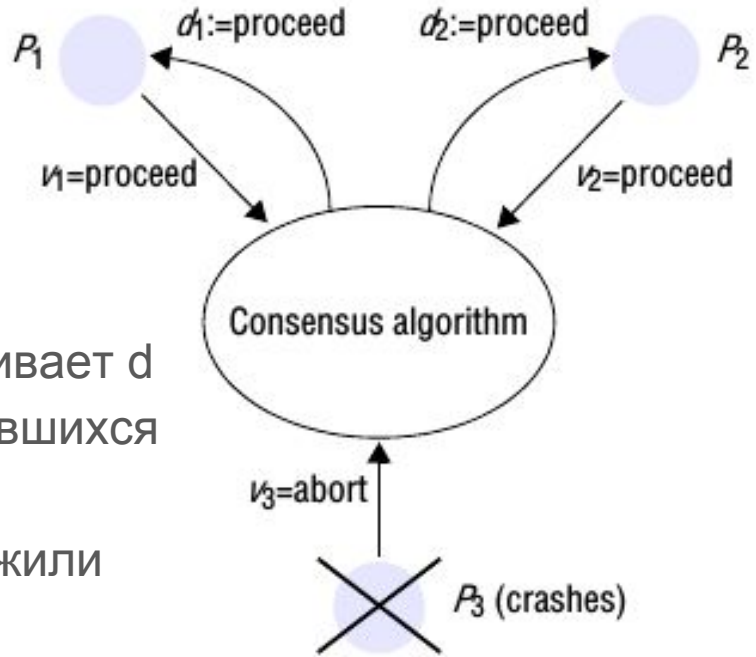
- E2 достигается за счёт надёжности каналов связи
- E1 достигается, если процессы не заменяются
  - проблема, если вместо убитого процесса появляется новый с уже имеющимся id
  - проблема, если таймауты подобраны неверно
- $O(N^2)$ сообщений в худшем случае

Проблемы соглашения

- задача договориться о каком-то значении, которое было предложено одним или несколькими процессами
- процессы взаимодействуют посылкой сообщений
- каналы связи надёжны
- возможны отказы процессов

Задача консенсуса

- состояния
  - не определился
  - определился
- переменные
  - v -- предложенное значение
  - d -- зафиксированное значение
- T: каждый корректный процесс устанавливает d
- A: значения d всех корректных определившихся равны
- I: если все корректные процессы предложили одно значение, любой корректный и определившийся имеет это значение в d



Консенсус в синхронных системах

Algorithm for process $p_i \in g$; algorithm proceeds in $f+1$ rounds

On initialization

$Values_i^1 := \{v_i\}$; $Values_i^0 = \{\}$;

In round r $(1 \le r \le f+1)$

B-multicast$(g,\ Values_i^r - Values_i^{r-1})$; // Send only values that have not been sent

$Values_i^{r+1} := Values_i^r$;

while (in round r)

{

On B-deliver(V_j) *from some* p_j

$Values_i^{r+1} := Values_i^{r+1} \cup V_j$;

}

After $(f+1)$ *rounds*

Assign d_i = *minimum*$(Values_i^{f+1})$;

Особенности

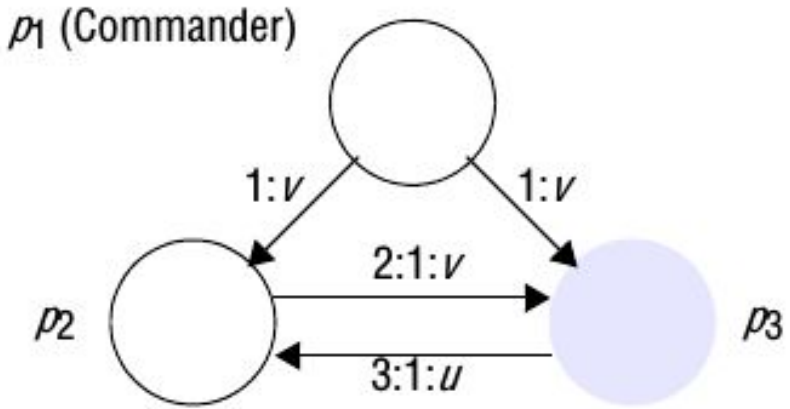
- если процессы не отказывают, задача решается мультикастом
- если процессы умирают, то процесс может не завершиться
  - особенно если система асинхронная
- если допустимы злоумышленники, процессы должны сверять значения

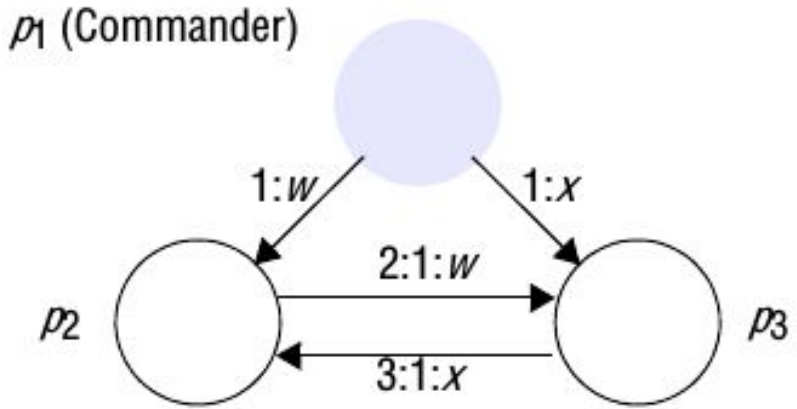
Задача византийских генералов

- один процесс предоставляет решение, о котором остальные должны договориться
- ряд процессов может быть злоумышленниками
- каналы передачи данных закрытые
- T: каждый корректный процесс устанавливает d
- A: значения d всех корректных определившихся равны
- I: если командир корректен, все остальные корректные процессы соглашаются с его решением

Решение для синхронных систем

- невозможно для N <= 3f, если сообщения не подписываются





Решение для N >= 3f + 1

1. командир шлёт сообщения лейтенантам
2. лейтенанты обмениваются полученными сообщениями