Распределенные вычислительные системы

Обмен данными
CAP-теорема

Вопросы

- Как расшифровывается и в чем состоит принцип архитектуры NUMA?
- Каким образом использование конвейерных вычислений ускоряет решение научных задач?
- Назовите основные отличия РВС от систем с общей памятью.
- Назовите основные виды прозрачности РВС.

Обмен данными

Маршализация

- Для передачи параметров по сети используется **маршализация** (marshalling) - процесс преобразования параметров для передачи их между процессами при удаленном вызове
- Маршализация
  - **по ссылке** – экземпляр удаленного объекта находится на сервере и не покидает его, а для доступа используются посредники
  - **по значению** – удаленный объект сериализируется и его копия передается в другой процес

Форматы сериализации данных

- Текстовые (платформо-независимые):
  - XML
  - JSON
  - YAML
- Бинарные
  - Protocol Buffers (Google)
  - Byte stream (очень неэффективные):
    - java.io.Serializable interface
    - .NET Serializable attribute
  - MessagePack

XML vs JSON

XML

```
<person>
  <firstName>Иван</firstName>
  <lastName>Иванов</lastName>
  <address>
    <streetAddress>Московское ш., 101,
кв.101</streetAddress>
    <city>Ленинград</city>
    <postalCode>101101</postalCode>
  </address>
  <phoneNumbers>
    <phoneNumber>812 123-
1234</phoneNumber>
    <phoneNumber>916 123-
4567</phoneNumber>
  </phoneNumbers>
</person>
```

363 символа

JSON

```
{
   "firstName": "Иван",
   "lastName": "Иванов",
   "address":  {
      "streetAddress": "Московское ш.,
101, кв.101",
      "city": "Ленинград",
      "postalCode": 101101
   },
   "phoneNumbers":  [
      "812 123-1234",
      "916 123-4567"
   ]
}
```

316 символов

Google Protocol Buffers

- Protocol Buffers — язык описания данных, предложенный Google в 2008 году, как альтернатива XML. Предполагается, что Protocol Buffers проще и легче, чем XML.
- Сначала должна быть описана структура данных, которая затем компилируется в классы, представляющие эти структуры. Вместе с классами идет код их сериализации в компактный формат представления.
- Примечательно, что можно добавлять к уже созданной структуре данных новые поля без потери совместимости с предыдущей версией: при анализе старых записей новые поля просто будут игнорироваться.
- PS: Используется в Diablo 3 ;0)

PROTOBUF

```
message Car {
    required string model = 1;

    enum BodyType {
        sedan = 0;
        hatchback = 1;
        SUV = 2;
    }

    required BodyType type = 2 [default = sedan];
    optional string color = 3;
    required int32 year = 4;

    message Owner {
        required string name = 1;
        required string lastName = 2;
        required int64 driverLicense = 3;
    }

    repeated Owner previousOwner = 5;
}
```

.proto - файл

MessagePack

- MessagePack – это бинарный формат предоставления данных, структура которого напоминает JSON (только более быстрый и более компактный).
- Был спроектирован, чтобы обеспечивать прозрачную конвертацию в JSON

JSON vs MessagePack

	JSON		MessagePack	
null	null	4 bytes	c0	1 byte
Integer	10	2 bytes	0a	1 byte
Array	[20]	4 bytes	91 14	2 bytes
String	”30”	4 bytes	a2 ‘3’	3 bytes
Map	{“40”:n	11 bytes	81 a1	5 bytes

Architecture of MessagePack by Sadayuki Furuhashi
http://www.slideshare.net/frsyuki/architecture-of-messagepack

MessagePack

JSON

```
{
   "firstName": "Иван",
   "lastName": "Иванов",
   "address":  {
      "streetAddress": "Московское ш.,
101, кв.101",
      "city": "Ленинград",
      "postalCode": 101101
   },
   "phoneNumbers":  [
      "812 123-1234",
      "916 123-4567"
   ]
}
```

338 bytes

84 a9 66 69 72 73 74 4e 61 6d 65 a8 d0 98 d0
b2 d0 b0 d0 bd a8 6c 61 73 74 4e 61 6d 65 ac
d0 98 d0 b2 d0 b0 d0 bd d0 be d0 b2 a7 61 64
64 72 65 73 73 83 ad 73 74 72 65 65 74 41 64
64 72 65 73 73 da 00 27 d0 9c d0 be d1 81 d0
88 2e 2c 20 31 30 31 2c 20 d0 ba d0 b2 2e 31
30 31 a7 63 69 74 79 aa d0 9b d0 b5 d0 bd d0
b8 d0 bd d0 b3 d1 80 d0 b0 d0 b4 aa 70 6f 73
74 61 6c 43 6f 64 65 ce 00 01 8a ed ac 70 68 6f
6e 65 4e 75 6d 62 65 72 73 92 ac 38 31 32 20
31 32 33 2d 31 32 33 34 ac 39 31 36 20 31 32
33 2d 34 35 36 37

MessagePack (hex)
187 bytes 55 %

http://msgpack.org/

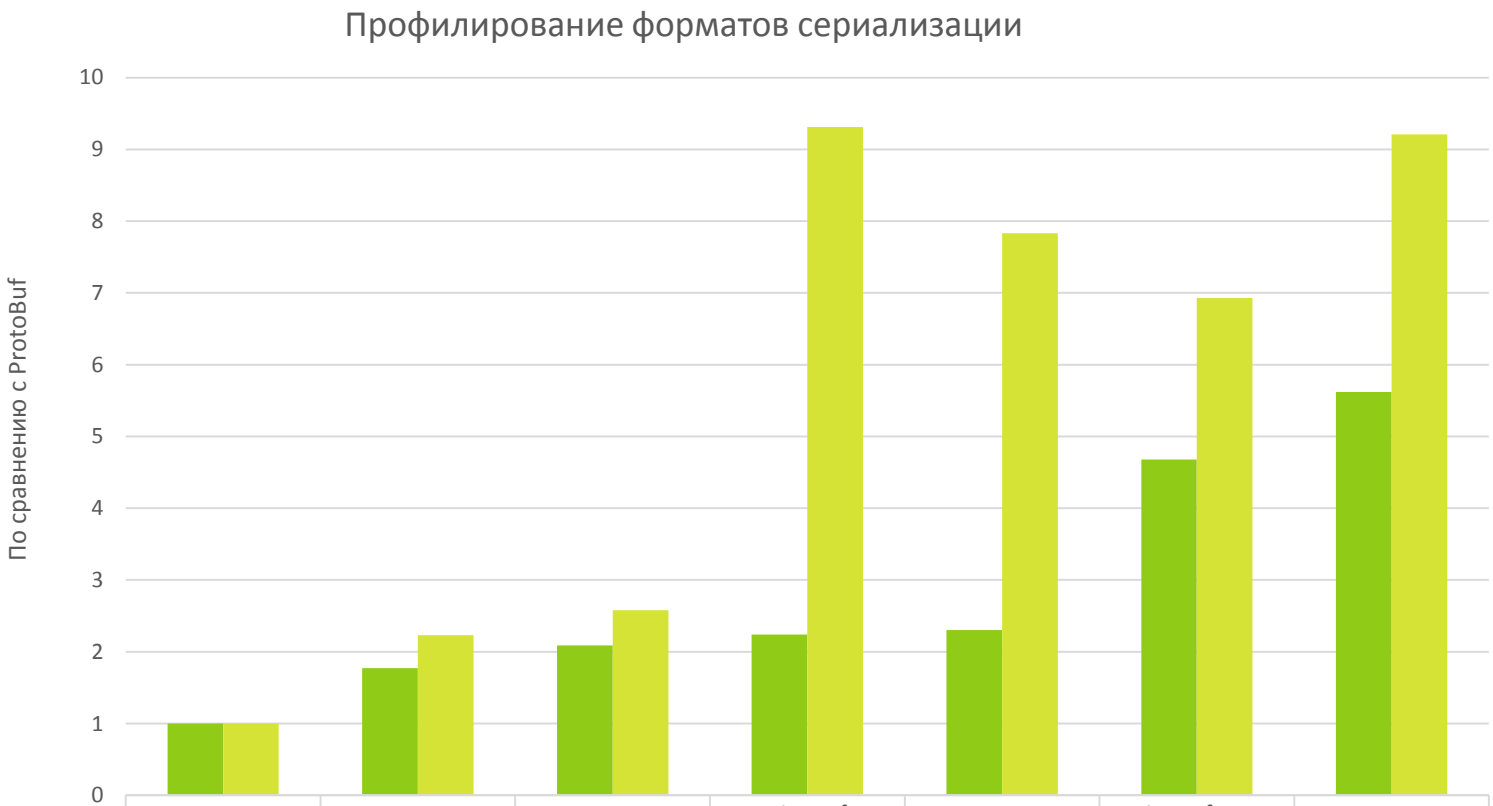
MessagePack

- JSON+ZIP VS MessagePack:
  - На 50% падает производительность при архивировании / разархивировании
  - Невозможно работать с данными в режиме потока (напрямую), необходима обязательная предварительная разархивация

Форматы обмена данными

- JSON
  - человеко-читаемый / редактируемый
  - может быть разобран без предварительного знания схемы
  - отличная поддержка браузеров (JavaScript native class description)
  - компактнее, чем XML
- XML
  - человеко-читаемый / редактируемый
  - может быть разобран без предварительного знания схемы
  - стандарт для многих протоколов (SOAP и т.п.)
  - хорошая инструментальная поддержка (XSD, XSLT, SAX, DOM и т.д.)
  - не компактный, большой объем «лишних» описаний
- Protobuf (Google), MessagePack
  - очень компактный (плотная упаковка данных)
  - очень быстрая обработка
  - не предназначен для чтения/редактирования человеком
- Protobuf (Google):
  - встроенная поддержка версий протокола (при изменении протокола, клиенты могут работать со старой версией, пока не обновятся)
  - без знания схемы очень тяжело разобрать (бинарный формат данных, внутренне не однозначен, требуется схема для разбора)

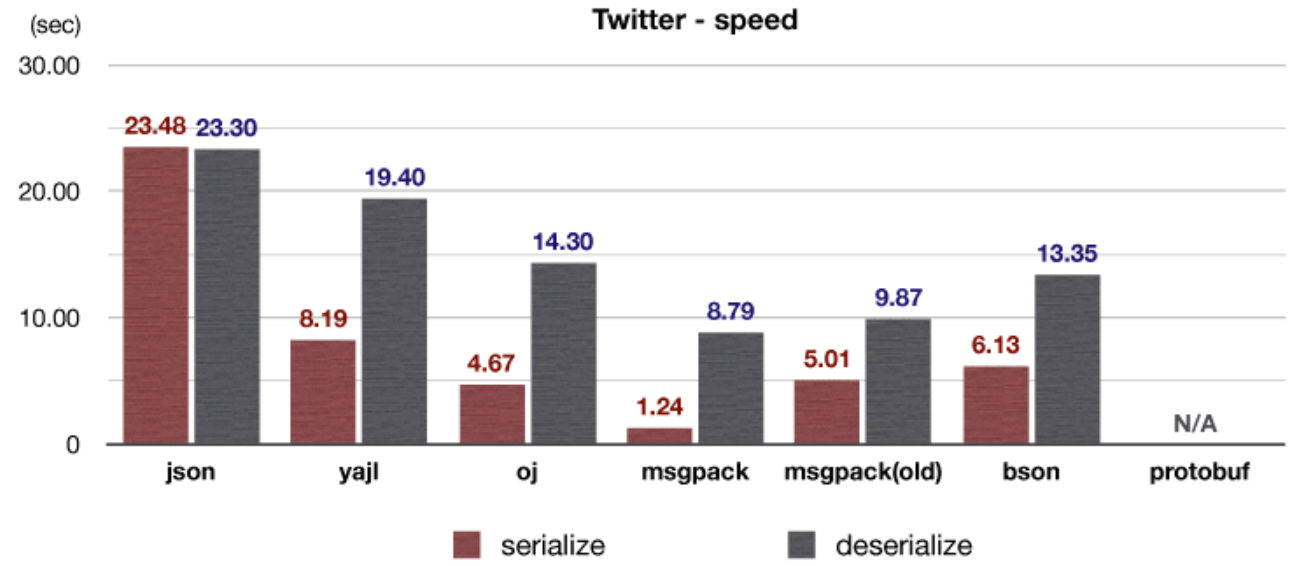
Форматы сериализации

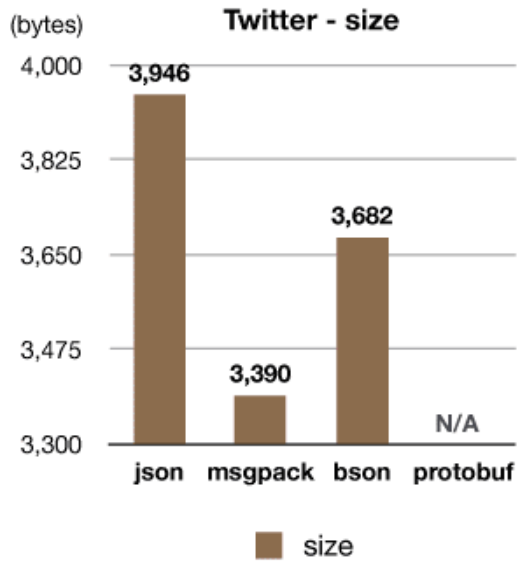


	ProtoBuf.net	ServiceStack TypeSerializer	ServiceStack JsonSerializer	Microsoft JsonDataContract Serializer	NewtonSoft.Json	Microsoft XML DataContractSeria lizer	Microsoft BinaryFormatter
Больше (раз)	1	1,77	2,09	2,24	2,3	4,68	5,62
Медленнее (раз)	1	2,23	2,58	9,31	7,83	6,93	9,21

Basically protocol buffers (protobuf-net) is around **7x** quicker than the fastest Base class library Serializer in .NET (XML DataContractSerializer). Its also smaller than the competition as it is also **2.2x** smaller than Microsoft's most compact serialization format (JsonDataContractSerializer).

http://www.servicestack.net/benchmarks/NorthwindDatabaseRowsSerialization.100000-times.2010-08-17.html





Adam Leonard. MessagePack for Ruby version 5
https://gist.github.com/adamjleonard/5274733

Когда использовать какие форматы?

- **XML**
  - Если система предоставляет публичный API в виде XML Веб-сервиса (изучим их позже)
  - Работаете с «классической» системой, в которой уже используется XML в качестве стандарта обмена данными
  - Требуются стандартные инструменты верификации и трансформации (например, в HTML) (XSD, XSLT, SAX, DOM и т.д.)
- **JSON**
  - Если система предоставляет публичный API в виде REST-сервиса (изучим их позже)
  - Если клиенты, в большинстве своем, реализуются на JavaScript
  - Более компактный чем XML (в 2-2.5 раза) и самый простой для чтения/редактирования – соответственно, везде, где вы хотели бы использовать XML, подумайте, может быть стоит использовать JSON.
- **MessagePack** – если требуется высокая скорость и компактность, совместимость с JSON, но не требуется редактирование/чтение человеком
- **Protobuf**
  - Если для вас прежде всего важен объем и скорость обработки данных
  - Если важна возможность простого обновления протокола
  - Если реализуется внутренний (не публичный) протокол обмена

CAP-ТЕОРЕМА

Управление Данными в PBC

В любой сетевой системе, *обеспечивающей хранение совместно доступных данных*, разработчики хотят поддерживать следующие свойства:

- **Согласованность данных (Consistency)** – в системе существует единственная версия данных, соответствующая последней по времени операции обновления.
- **Доступность данных (Availability)** – запрос к PBC в любой момент времени должен завершиться корректным откликом, не зависимо от того, к какому серверу производится подключение.
- **Устойчивость к разделению (Partition tolerance)** - расщепление распределённой системы на несколько изолированных секций не приводит к некорректности отклика от каждой из секций.

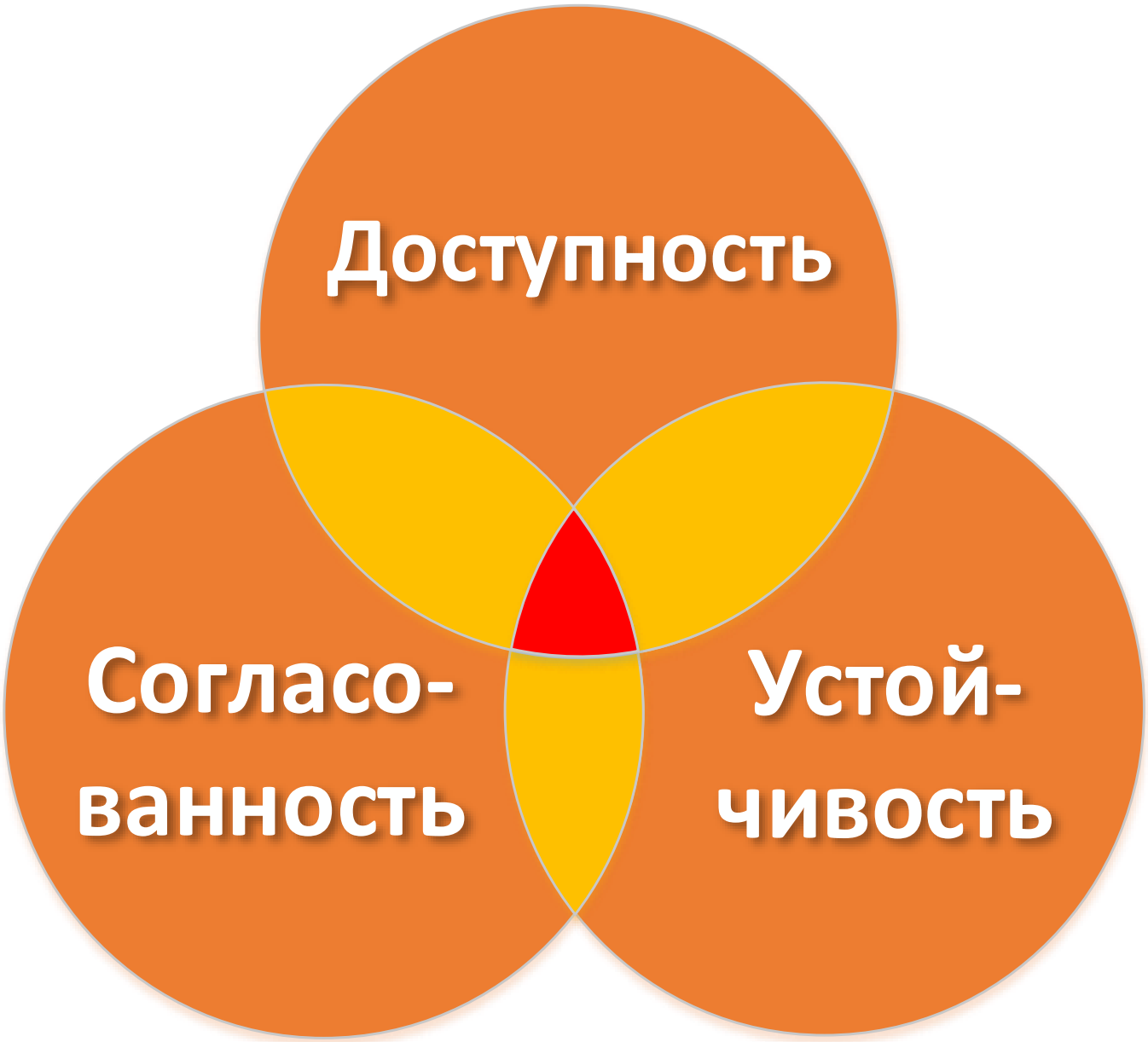
CAP-теорема

В 2000-м году Эрик Брюер (Eric Brewer – проф. Калифорнийского университета) предложил следующую теорему:

В любой сетевой системе, *обеспечивающей хранение совместно доступных данных*, одновременно могут поддерживаться только **два** из следующих трех свойств:

- **Согласованность данных (Consistency)**
- **Доступность данных (Availability)**
- **Устойчивость к разделению (Partition tolerance)**

В 2002-м году теорема была математически доказана в условиях отсутствия синхронизации (общих часов).


Доступность
Согласо-
ванность
Устой-
чивость

Что же такое Partition?

Разделение PBC это не только длительный разрыв между серверами, при котором один из них не может связаться с другим

» *Латентность сети также является разделением PBC.*

- Предположим, что у нас есть 2 сервера баз данных: один в России, другой в США.
- Они настроены на полное реплицированные
- Данные обновились на сервере в России. Через какое время сервер в США узнает об этом?
- **200 миллисекунд – лучший возможный результат** (худший возможный результат – никогда не узнает)
- В это «окно» они разделены – таким образом разделение системы происходит постоянно, даже в нормальных условиях работы PBC

Выбираем ли мы устойчивость?

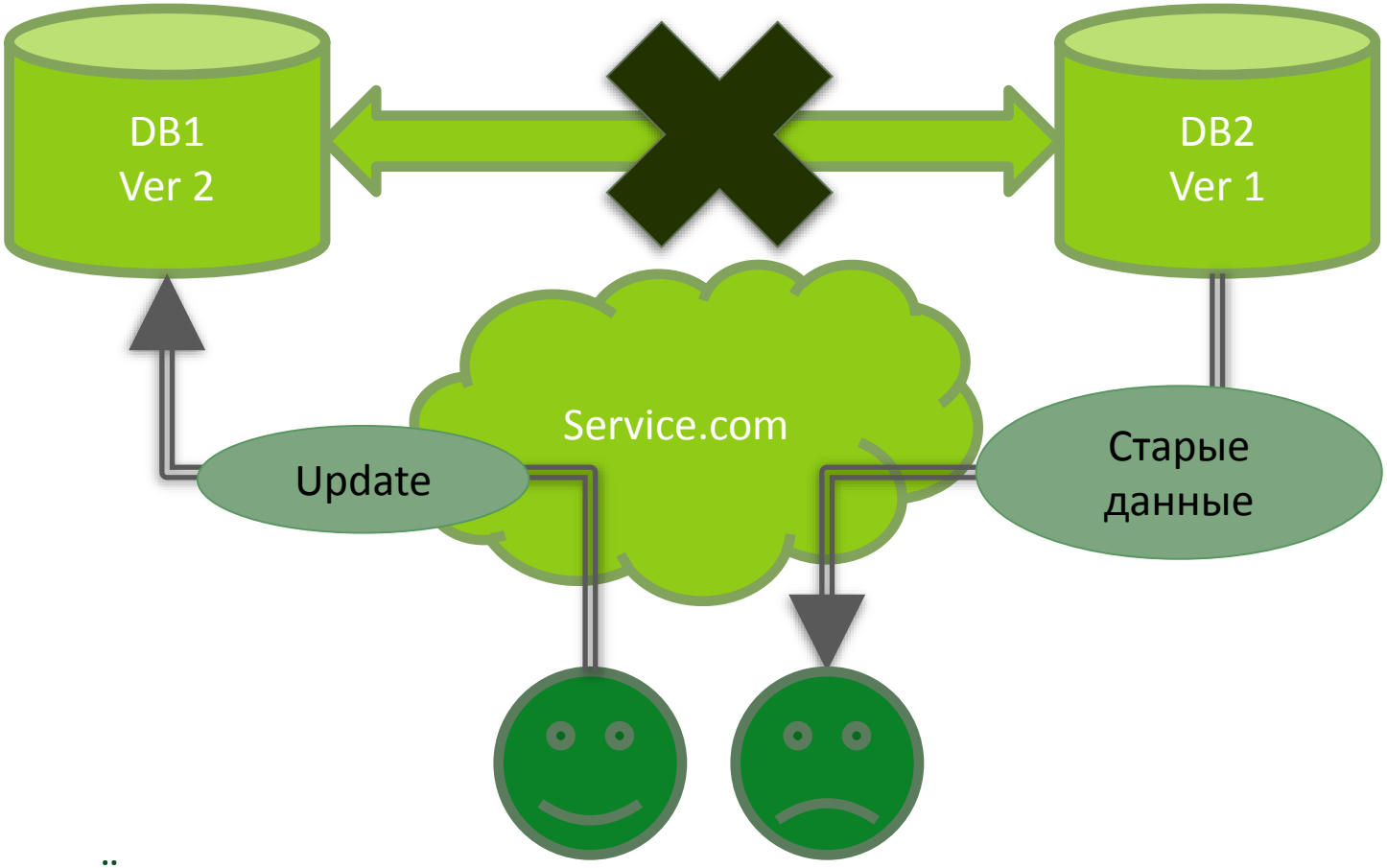
В реальном мире мы не можем выбирать, будут у нас сбои или нет. В РВС всегда будут возникать неполадки, зависит это от нас или нет:

- сетевые сбои,
- сбои работы оборудования,
- ошибки администрирования.

Плюс, любая латентность также вызывает разделение.

Поэтому приходится выбирать из двух вариантов:

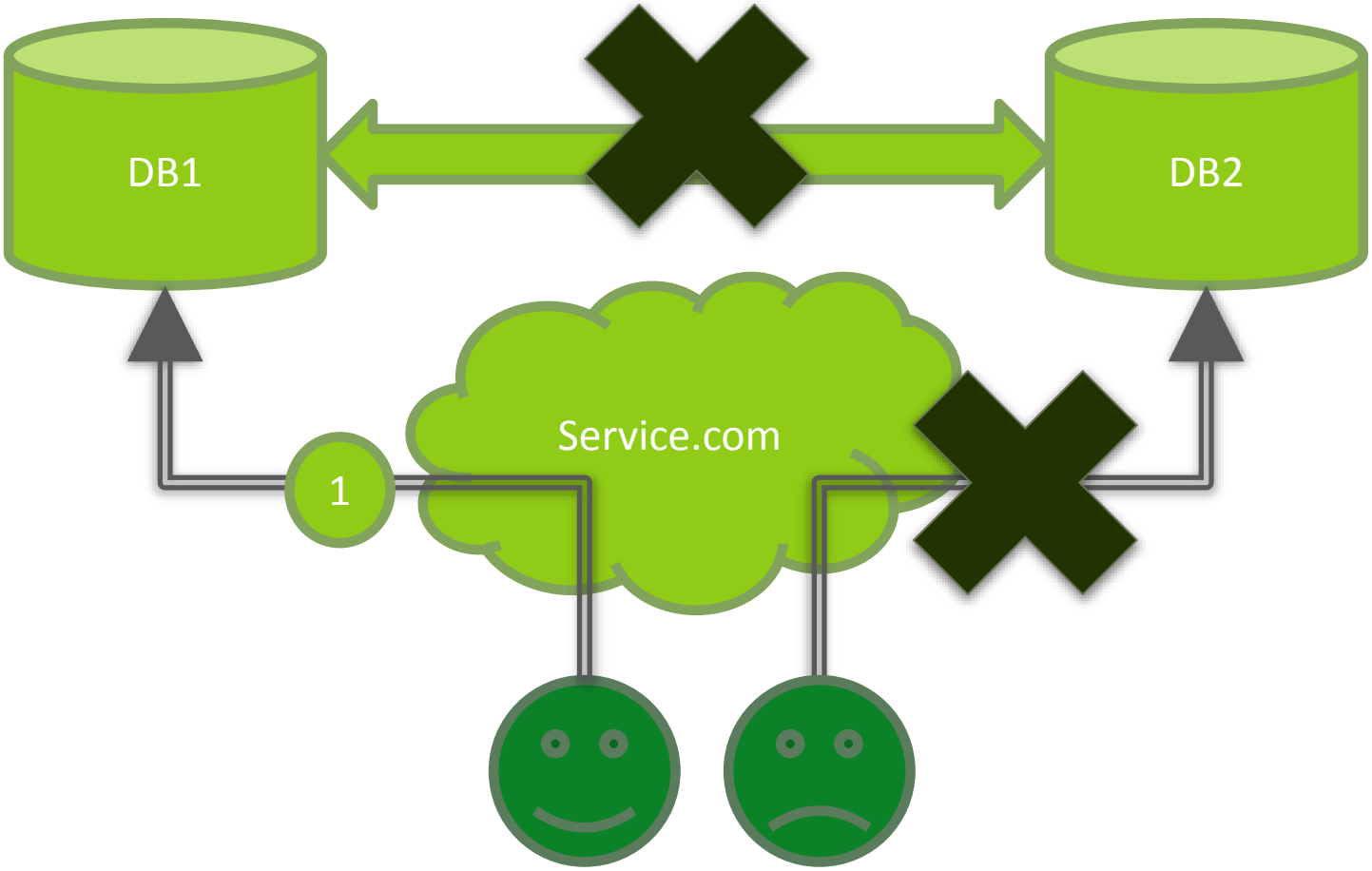
AP: 100% доступность, но несогласованность данных:



Распределённая система, отказывающаяся от целостности результата. Большинство NoSQL-систем принципиально не гарантируют целостности данных («целостные в конечном итоге» - eventually consistent).

CAP

CP: 100% согласованность, но недоступность данных при распаде:

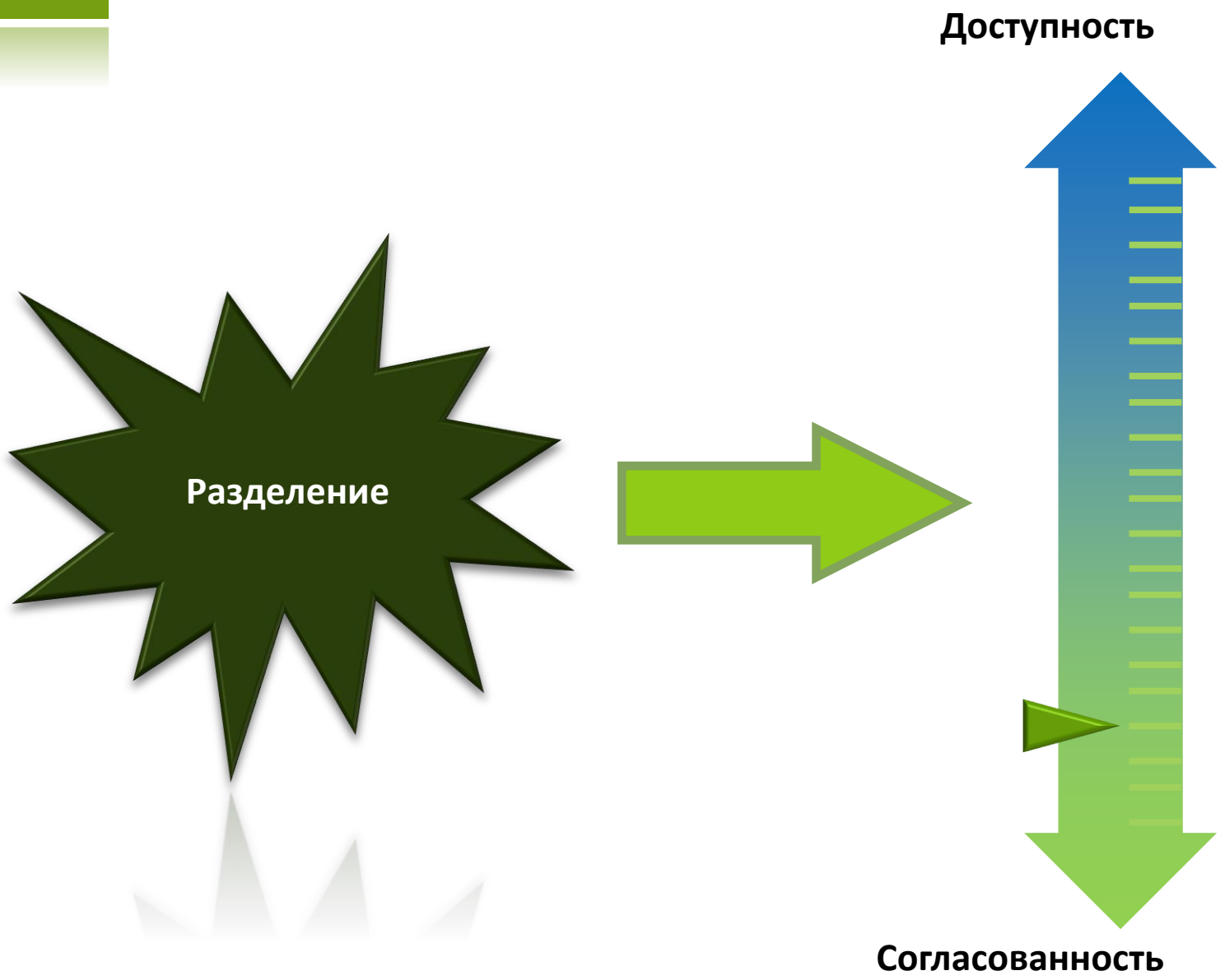


Распределённая система, в каждый момент обеспечивающая целостный результат и способная функционировать в условиях распада, в ущерб доступности может не выдавать отклик. Пессимистические блокировки для сохранения целостности.

Выбор между C и A

- Нет 100% работающего универсального решения, каким путем решать задачу обработки запросов при возникновении расщепления распределенной системы – выбирать доступность или же выбирать
- Этот выбор может быть сделан только разработчиком, на основе анализа **бизнес-модели** приложения.
- Более того, этот выбор может быть сделан не для всего приложения в целом, а отдельно для каждой его части:
  - *Согласованность* для процедуры покупки и оплаты в интернет-магазине;
  - *Доступность* для процедуры просмотра каталога товаров, чтения отзывов и т.п.
- Или же смириться с несогласованностью, в случае коротких периодов (1-2 минуты) разделения системы

Доступность и согласованность



Исключение из CAP- теоремы?

Системы без внутреннего состояния (STATELESS DISTRIBUTED SYSTEMS)

Eventual Consistency

В связи с невозможностью 100% времени поддерживать согласованность и доступность системы, пришлось искать компромисс.

- *Согласованность в конечном счете (eventual consistency) или слабая согласованность (weak consistency)* - означает, что если в течение достаточно долгого периода времени в систему не поступают новые операции обновления данных, то можно ожидать, что результаты всех предыдущих операций обновления данных в конце концов распространятся по всем узлам системы, и все реплики данных **согласуются**.
- При отсутствии сбоев, максимальный размер окна несогласованности может быть определен на основании таких факторов, как задержка связи, загруженность системы и количество реплик в соответствии со схемой репликации.
- Самая популярная система, реализующая «согласованность в конечном счете» – DNS. Обновленная запись распространяется в соответствии с параметрами конфигурации и настройками интервалов кэшированя. В конечном счете, все клиенты увидят обновление.

Более подробно о согласованности в конечном итоге: http://habrahabr.ru/post/100891/