Распределенные вычислительные системы

REST, CAP-теоремы

Версии REST-API

Версии API

- Когда вы делаете обновления, вы практически всегда вносите изменения во внутренние структуры данных и в модели.
- Версионирование включает в себя изменения в структуре ресурсов (например, добавление или удаление столбцов в базе данных).
- Если бы мы жили в идеальном мире, все клиенты автоматически бы обновлялись до новых версий протокола. Но это не так в реальном мире. Приходится одновременно поддерживать несколько версий.

Разделение с использованием URL

- api.example.com/v1/feeds будет использоваться версией 1 приложения
- api.example.com/v2/feeds будет использоваться версией 2.

- Несмотря на то, что звучит это все неплохо, это приводит к чрезмерному количеству URL для каждого изменения в формате возвращаемых данных.
- Такой подход рекомендуется для использования только в случае глобальных изменений в API.

Разделение с использованием URL

- api.example.com/v1/feeds будет использоваться версией 1 приложения
- api.example.com/v2/feeds будет использоваться версией 2.

- Несмотря на то, что звучит это все неплохо, это приводит к чрезмерному количеству URL для каждого изменения в формате возвращаемых данных.
- Такой подход рекомендуется для использования только в случае глобальных изменений в API.

Разделение на основе модели (структуры данных)

- При каждом изменении структуры данных необходимо изменять версию протокола.
- Идентификатор протокола можно указать в поле “UserAgent” в поле HTTP-запроса:

```
Request
/login
Headers
Authorization: Token XXXXX
User-Agent: MyGreatApp /1.0
Accept: application/json
Accept-Encoding: compress, gzip
Parameters Encoding type – application/x-www-form-urlencoded
token – “Facebook Auth Token” (mandatory) profileInfo = “json string
containing public profile information from Facebook” (optional)
```

Разделение на основе модели (структуры данных)

- В этом случае, логика создания объекта-обработчика в зависимости от протокола может быть реализована на основе паттерна «Фабричный метод»:

```
Feed myFeedObject = Feed.createFeedObject("1.0");
myFeedObject.populateWithDBObject(FeedDao* feedDaoObject);
```

- Внесение изменений не будет нарушать имеющиеся соглашения. Просто создавайте новые структуры данных (модели), вносите изменения в «Фабричный метод» для создания экземпляра новой модели для новой версии и сразу несколько версий приложения могут работать одновременно с одним сервером.

Примеры REST API

Twitter REST API v1.1

`GET statuses/retweets/:id`

Вернет до 100 ретвитов твита с номером id

`GET statuses/show/:id`

Вернет отдельный твит «id»

`GET statuses/destroy/:id`

Удалить твит «id»

`GET statuses/update`

Обновить статус (создать новый твит)

Google Translate REST API

IN:

```
GET https://www.googleapis.com/language/translate/v2?
key=INSERT-YOUR-KEY&source=en&target=de&q=Hello%20world
```

OUT :

```
200 OK
{
	"data": {
		"translations": [
		{
			"translatedText": "Hallo Welt"
		}
		]
	}
}
```

PayPal REST API

IN: `https://api.paypal.com/v1/payments/payment`

curl -v https://api.sandbox.paypal. com/v1/payments/payment \ -H "Content-Type:application/json" \ -H "Authorization:Bearer EMxItHE7Zl4cMdkvMg-f7c63GQgYZU8FjyPWKQlpsqQP" \ -d '{ "intent":"sale", "payer": { "payment_method":"credit_card", "funding_instruments":[{ "credit_card": { "number":"4417119669820331", "type":"visa", "expire_month":11, "expire_year": 2018, "cvv2":"874", "first_name":"Joe", "last_name":"Shopper", "billing_address": {"line1":"52 N Main ST", "city":"Johnstown", "country_code":"US", "postal_code":"43210", "state":"OH" } } }] },

"transactions":[{ "amount":{ "total":"7.47", "currency":"USD", "details": { "subtotal":"7.41", "tax":"0.03", "shipping":"0.03" } }, "description":"This is the payment transaction description.” }] }'

PayPal REST API

OUT:

200 OK

{ "id": "PAY-17S8410768582940NKEE66EQ", "create_time": "2013-01-31T04:12:02Z", "update_time": "2013-01-31T04:12:04Z", "state": "approved", "intent": "sale", "payer": {

…

Разработка RESTful-сервиса

Разработка сервера

- Ruby on Rails
  - Много магии
- Java – JAX-RS
  - Очень популярный язык
  - Очень популярная платформа для сервисов
- Python – Django
  - Попробуете на лабораторной.

JAVA SERVICE

```
@Path("/stores")
public class StoreService {

@GET
@Produces("application/xml")
public JAXBElement <Stores> getStoresAsXML()    {
   Stores stores = Stores.getStores();
   return new JAXBElement <Stores>
      ( new Qname("Stores"), Stores.class, stores);
}

@Path("/{id}")
@GET
@Produces("application/xml")
public Store getStoreAsXML(@ PathParam("id") String id) {
   // implementation here
}
```

Daigneau, Robert. Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services. Pearson Education. Kindle Edition.

Java service

```
@POST
@Consumes("application/xml")
@Produces("application/xml")
public Store createStore(JAXBElement <Store> store)    {
   // implementation here
}

@Path("/{id}")
@PUT
@Produces("application/xml")
public Store updateStore(@PathParam("id") String id)   {
   // implementation here
} }
```

Daigneau, Robert. Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services. Pearson Education. Kindle Edition.

Java Client

Для создания клиента можно использовать пакет **Jersey**. Jersey предоставляет REST-клиент и REST-сервер.

```
public class Test {
public static void main(String[] args) throws ClientProtocolException,
IOException {
        Client client = Client.create();
        WebResource r = client.resource(“http://localhost:8080/xyz”);
        MultivaluedMap<String, String> params = new MultivaluedMapImpl();
        params.add("foo", "x");
        params.add("bar", "y");
        // getting XML data: http://localhost:8080/xyz/abc?foo=x&bar=y
        System.out.println(r.path(“abc”).
queryParams(params).accept(MediaType.APPLICATION_XML).get(String.class));
        // getting JSON data: http://localhost:8080/xyz/abc?foo=x&bar=y
        System.out.println(r.path(“abc”).
queryParams(params).accept(MediaType.APPLICATION_JSON).get(String.class));
         }
}
```

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.

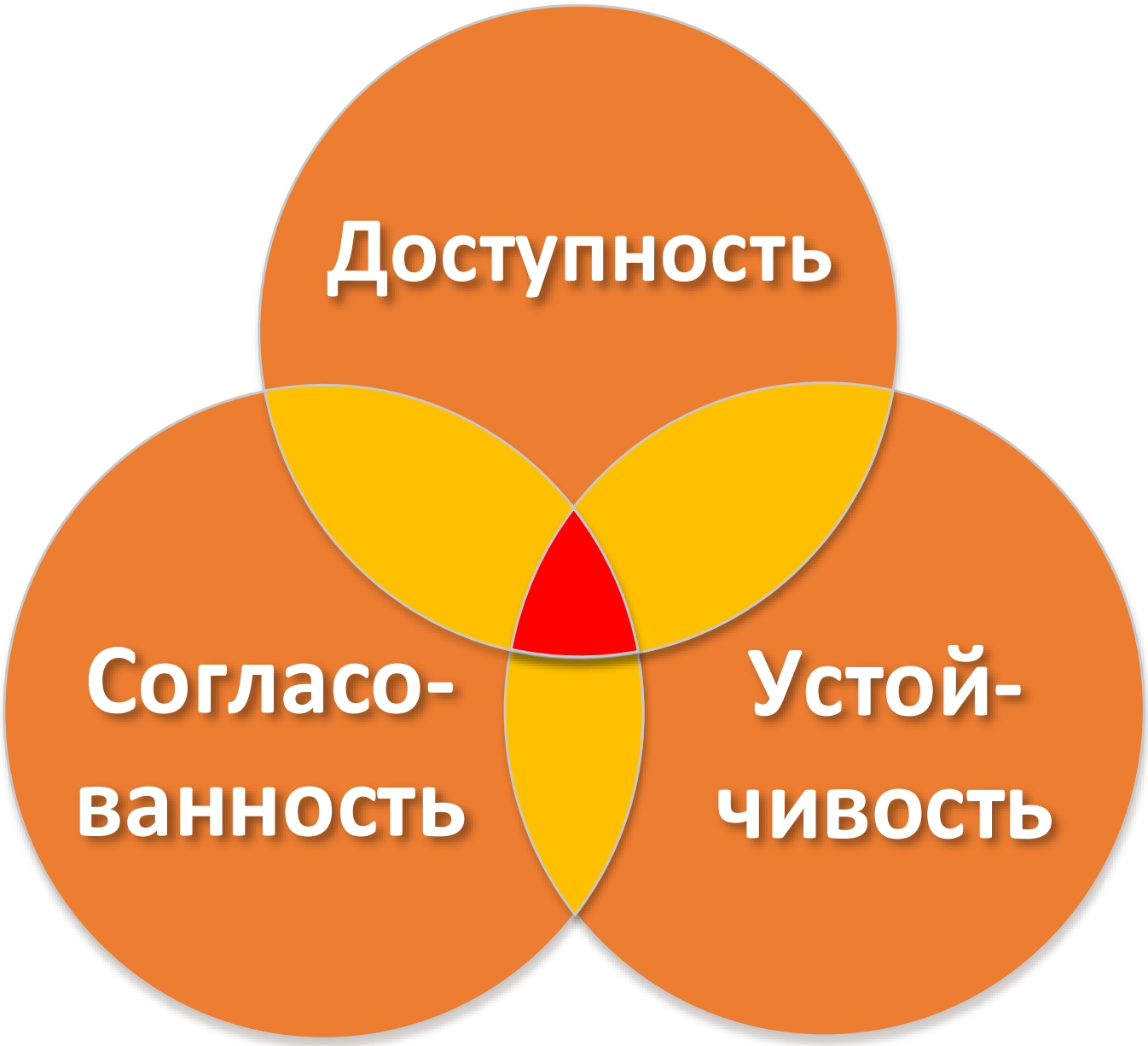
CAP-теорема

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?

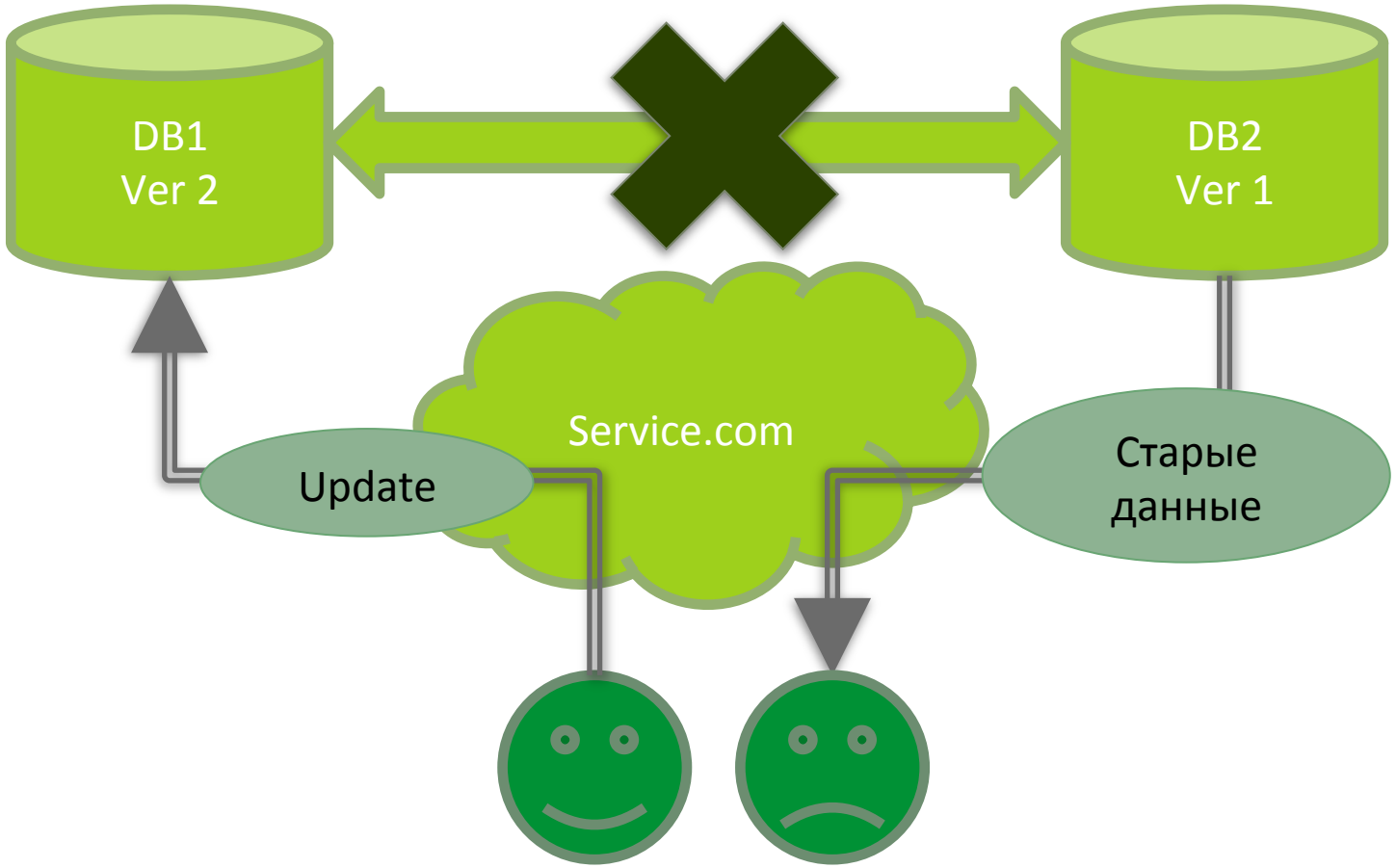
В реальном мире мы не можем выбирать, будут у нас сбои или нет. В PBC всегда будут возникать неполадки, зависит это от нас или нет:

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

Плюс, любая латентность также вызывает разделение.

Поэтому приходится выбирать из двух вариантов:

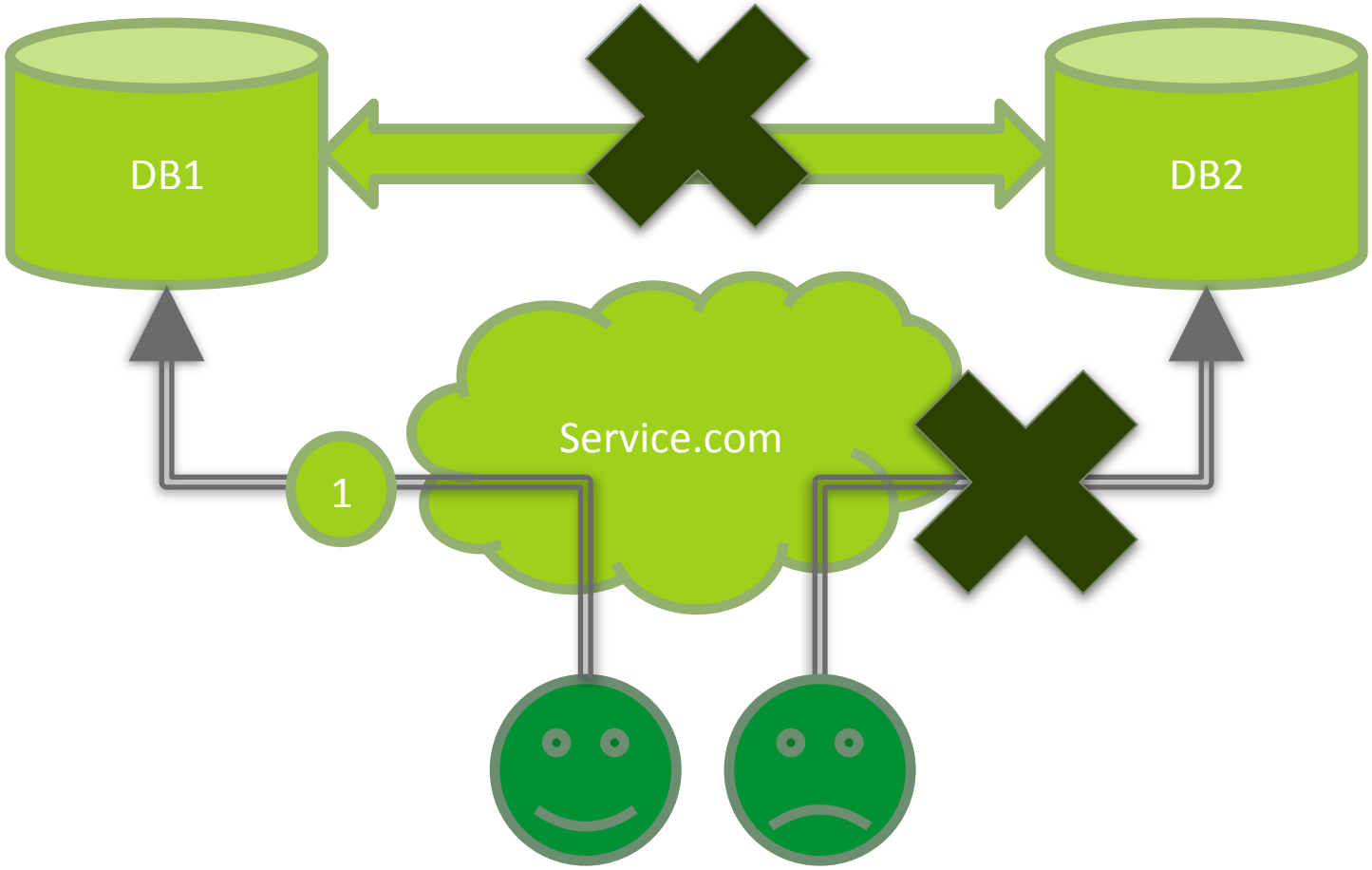
AP: 100% доступность, но несогласованность данных:



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:

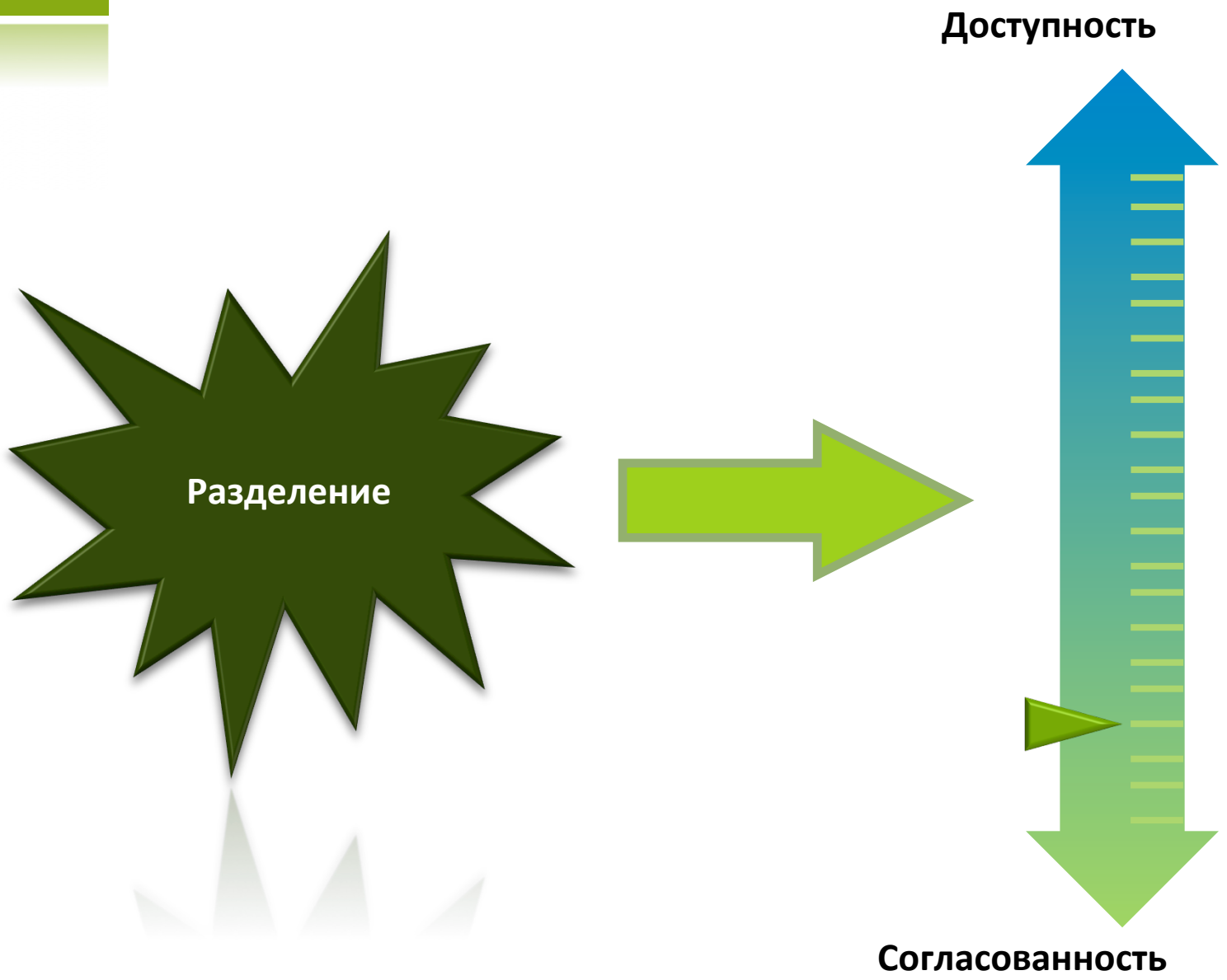


Распределённая система, в каждый момент обеспечивающая целостный результат и способная функционировать в условиях распада, в ущерб доступности может не выдавать отклик. Пессимистические блокировки для сохранения целостности.

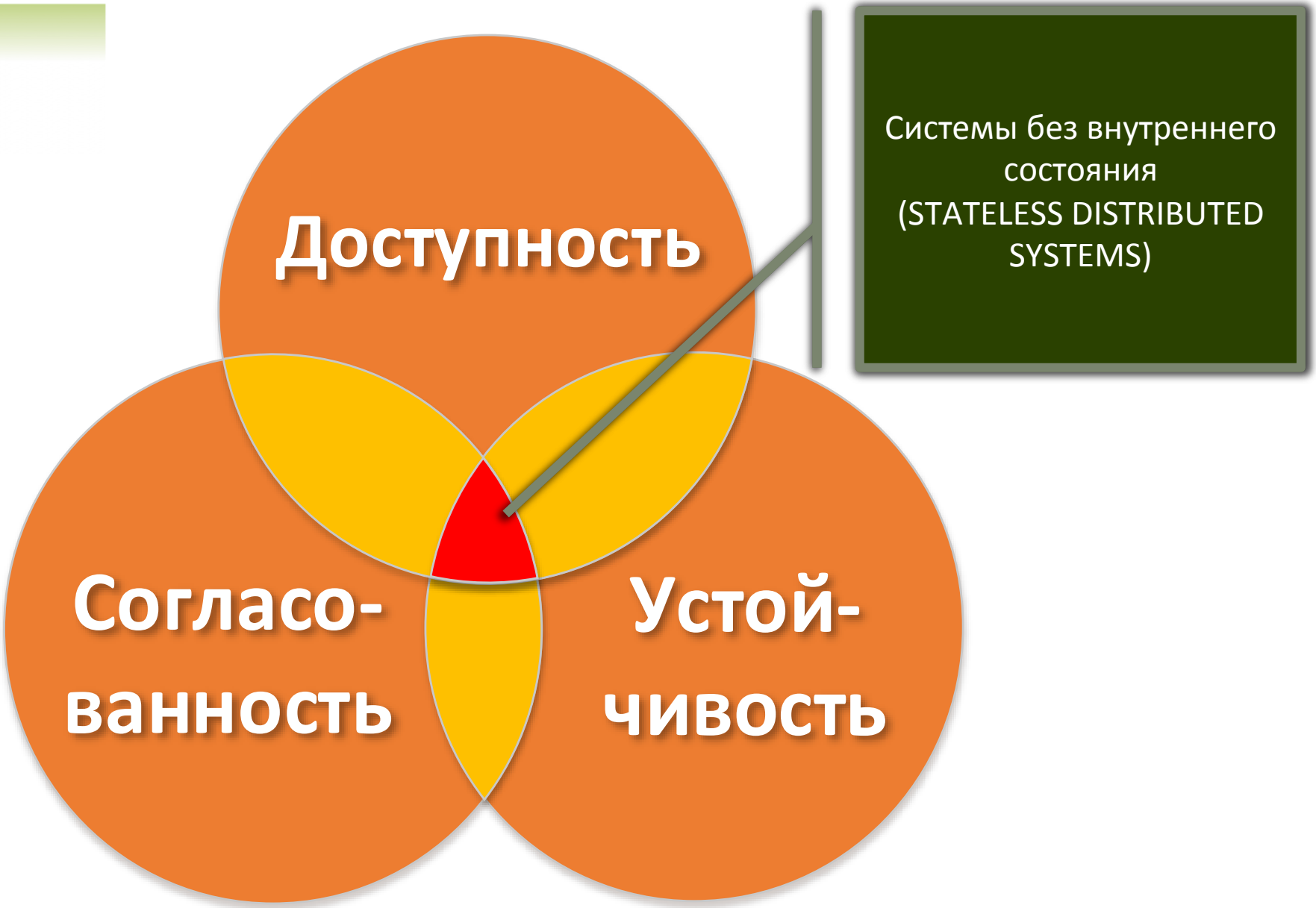
Выбор между C и A

- 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?



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/