Распределенные вычислительные системы

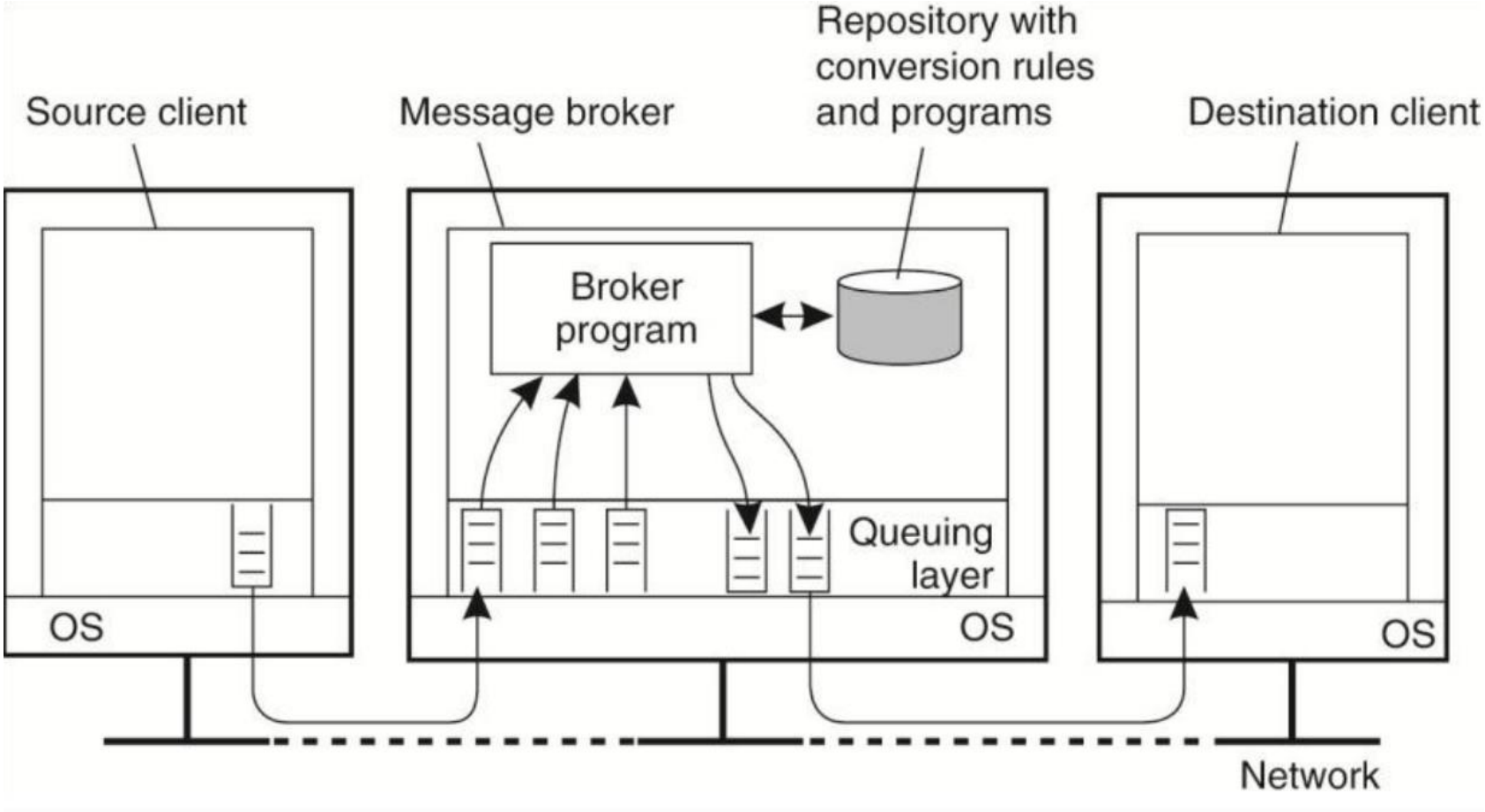
Менеджеры сообщений. Форматы данных

Взаимодействие посредством менеджера сообщений

Менеджеры (очереди) сообщений

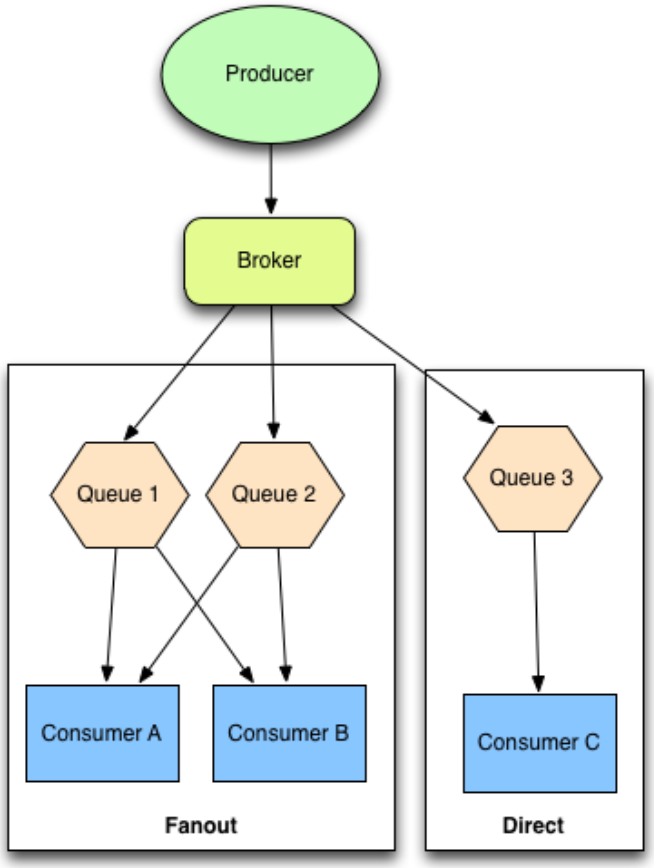
- *Очередь сообщений* – это ПО промежуточного слой, обеспечивающие управление взаимодействием между элементами РВС, посредством сбора, хранения и маршрутизации сообщений между процессами.
- Очередь сообщений позволяет работающим в различное время приложениям взаимодействовать в гетерогенных сетях и системах, которые могут временно отключаться от сети.
- Приложения отправляют, получают и считывают (то есть читают без удаления) сообщения из очередей.

Использование менеджера сообщений



Менеджеры (очереди) сообщений

- Очередь сообщения предоставляет интерфейс для *поставщика (producer)* и *потребителя (consumer)* сообщений.
- Менеджер сообщений может обеспечить распределение сообщений в различные очереди, обеспечивая распределение нагрузки между задачами и возможность горизонтального масштабирования.



Использование менеджера сообщений

Достоинства

- **Слабосвязанность** программных компонентов обеспечивается посредством единого интерфейса данных
- **Избыточность данных:** парадигма “put-get-delete” позволяет избежать потери данных, даже если они не были обработаны после их получения
- **Масштабируемость и эластичность:** считывать и обрабатывать заявки из очереди могут несколько независимых процессов одновременно. При этом, если один из таких процессов падает, любой другой может взять на себя задачу обработки сообщений
- **Очередность**: сообщения попадают в очередь одно за другим, и, соответственно, обрабатываются в порядке поступления.
- **Буферизация:** если время на обработку сообщения больше, чем время поступления новых сообщений, очередь буферизует новые сообщения и они не теряются.
- **Асинхронность взаимодействия:** время работы клиента и сервера не зависят друг от друга. Клиент может отправить сообщение и продолжить свою работу не дожидаясь ответа.
- **Высокая прозрачность:** вся коммуникация проходит через менеджер сообщений. Таким образом, можно в любой момент времени оценить, какие сообщения, от кого и кому поступали.

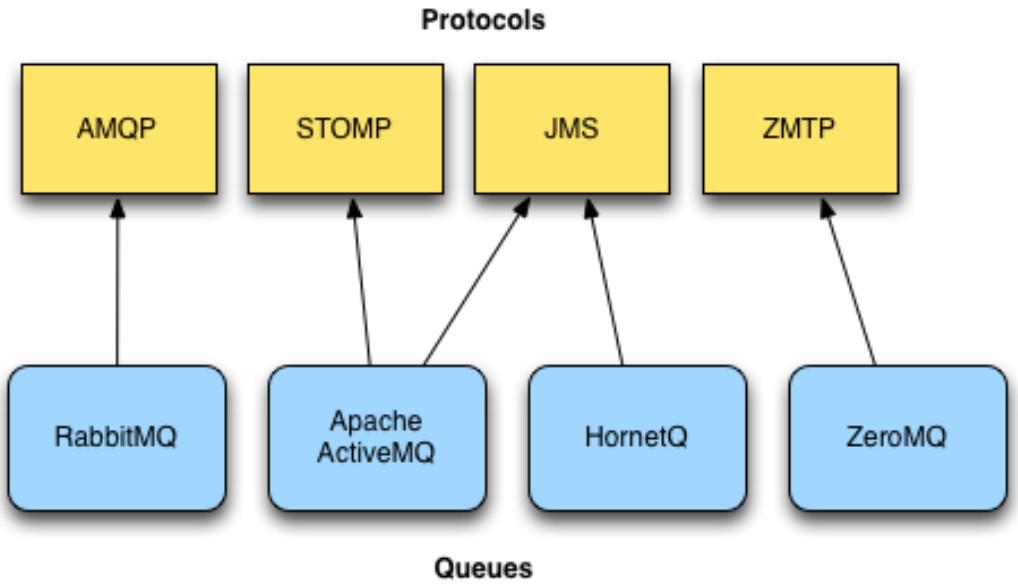
Использование менеджера сообщений

Недостатки

- необходимость явного использования очередей распределенным приложением;
- сложность реализации **синхронного** обмена;
- определенные накладные расходы на использование менеджеров очередей;
- сложность получения ответа: передача ответа может потребовать отдельной очереди на каждый компонент, посылающий заявки.

Службы очередей сообщений

- Службы очередей сообщений (Message Queue Services, MQS) используются в разработке начиная с 1980-х
- IBM WebSphere MQ (~8 000 $ на 100 процессоров).
- JMS – Java Message Service
- Microsoft Message Queuing (MSMQ - .NET)



RabbitMQ vs ActiveMQ

- RabbitMQ (http://www.rabbitmq.com/)
  - Создан на основе Erlang
  - Работает на всех основных операционных системах
  - Поддерживает огромное количество платформ для разработчиков (чаще всего – Python, PHP, Ruby)
  - Конфигурация – на основе Erlang
- Apache ActiveMQ (http://activemq.apache.org/)
  - Построен на основе Java Message Service
  - Чаще всего используется совместно с Java-стеком (Java, Scala, Clojure и др).
  - Также поддерживает STOMP (Ruby, PHP, Python).
  - Конфигурация – на основе XML

Пример использования RabbitMQ

- RabbitMQ as a Service: http://www.cloudamqp.com/
- Пример приложения: https://github.com/cloudamqp/java-amqp-example
- Запускаем Sender (OneOffProcess.java), потом Receiver (WorkerProcess.java)

Sender:

```
channel.queueDeclare(QUEUE_NAME, false, false, false, null);
String message = "Hello CloudAMQP!";
channel.basicPublish("", QUEUE_NAME, null, message.getBytes());
System.out.println(" [x] Sent '" + message + "'");
```

Receiver:

```
while (true) {
      QueueingConsumer.Delivery delivery = consumer.nextDelivery();
      String message = new String(delivery.getBody());
      System.out.println(" [x] Received '" + message + "'");
    }
```

Форматы сообщений

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)

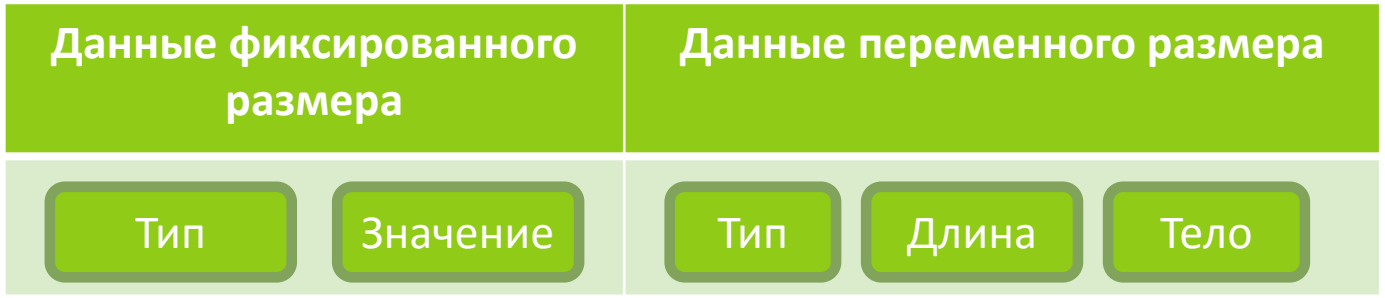
PROTOBUF

```
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 a…
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/

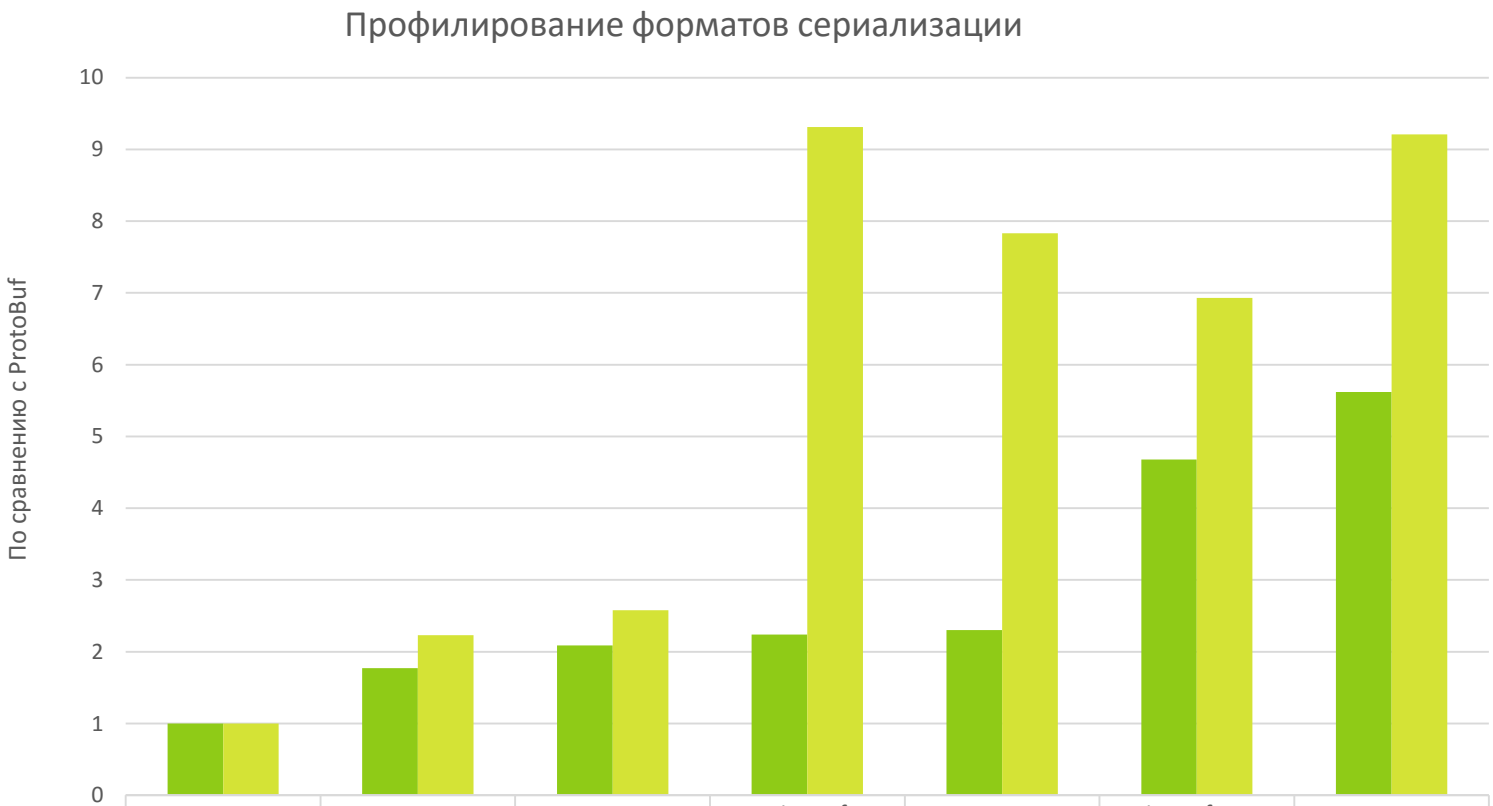
MessagePack

- 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)

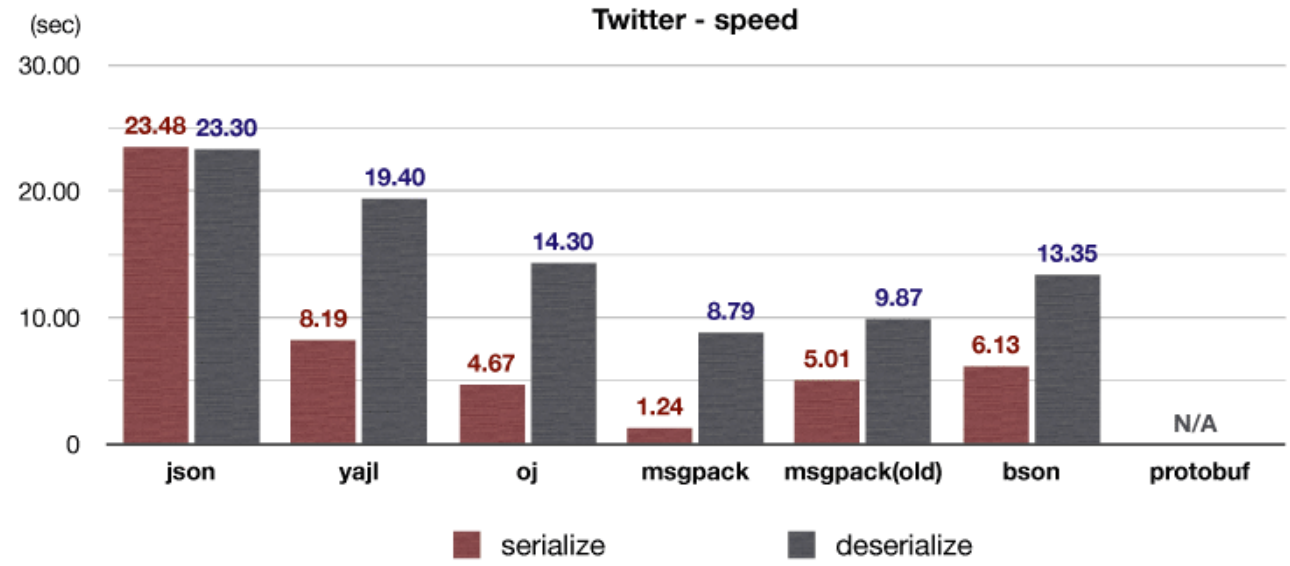
Форматы сериализации

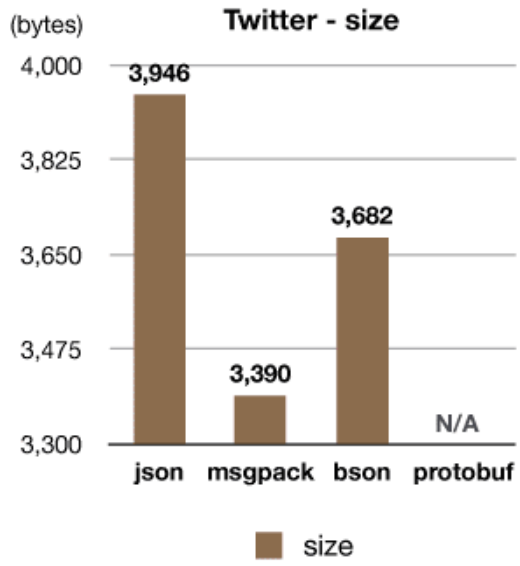


	ProtoBuf.net	ServiceStack TypeSerializer	ServiceStack JsonSerializer	Microsoft JsonDataContract Serializer	NewtonSoft.Json	Microsoft XML DataContractSeria lizer	Microsoft BinaryFormatter
Больше (раз)	1	1,77	2,09	2,24	2,3	4,68	5,62
Медленнее (раз)	1	2,23	2,58	9,31	7,83	6,93	9,21

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