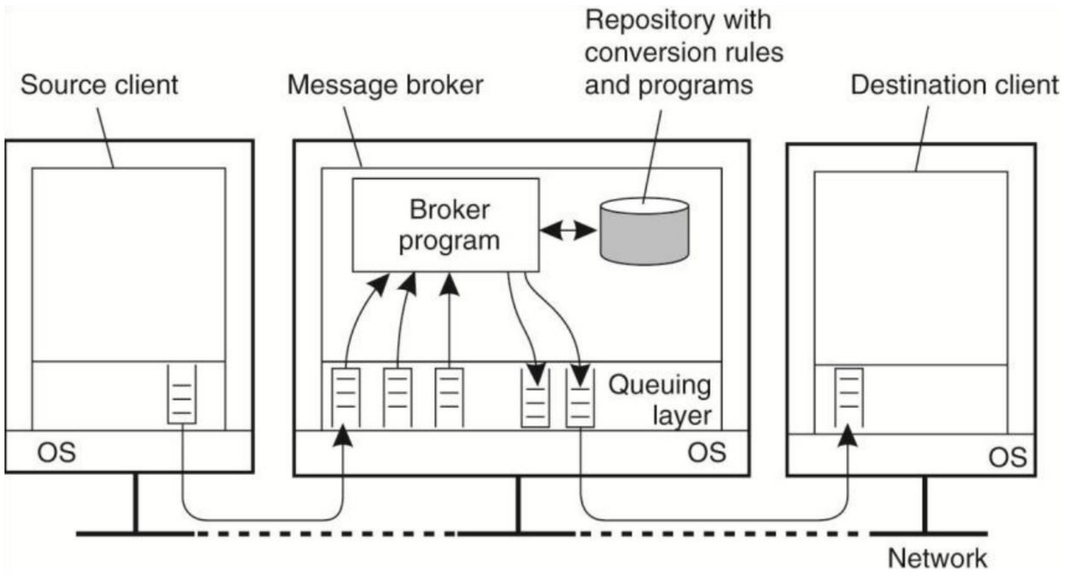
Distributed computing systems

Message Queues; Data Serialization

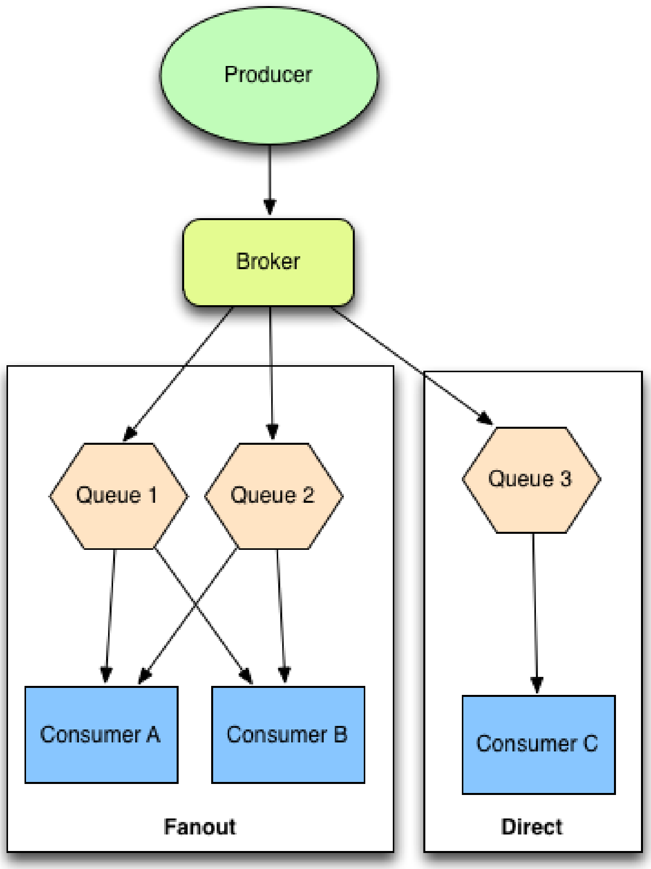
The Message System Paradigm

- The Message System or Message-Oriented Middleware (MOM) paradigm is an elaboration of the basic message-passing paradigm.
- In this paradigm, a message system serves as an intermediary among separate, independent processes.
- The message system acts as a switch for messages, through which processes exchange messages asynchronously, in a decoupled manner.
- A sender deposits a message with the message system, which forwards it to a message queue associated with each receiver. Once a message is sent, the sender is free to move on to other tasks.



Message Queues

- At the simplest level, a message queue is a way for applications and components to send messages between one another in order to reliably communicate.
- They connect **producers** which create messages and the **consumers** which then process them.
- Within the context of a web application, one common case is that the **producer** is a client application (i.e Rails or Sinatra) that creates messages based on interactions from the user (i.e user signing up).



Using Message Manager

The use of message queues is oriented towards **asynchronous** communication.

Advanteges

- **Decoupling:** By introducing a layer in between processes, message queues create an implicit, data-based interface that both processes implement.
- **Redundancy :** The put-get-delete paradigm, requires a process to explicitly indicate that it has finished processing a message before the message is removed from the queue.
- **Scalability** Because message queues decouple your processes, it's easy to scale up the rate with which messages are added to the queue or processed.
- **Elasticity & Spikability:** application needs to be able to keep functioning with this increased load, but the traffic is anomaly, not the standard. Message queues will allow beleaguered components to struggle through the increased load.
- **Resiliency**: Message queues decouple processes, so if a process that is processing messages from the queue fails, messages can still be added to the queue to be processed when the system recovers
- **Buffering:** Message queues help process-consuming tasks operate at peak efficiency by offering a buffer layer--the process writing to the queue can write as fast as it's able to, instead of being constrained by the readiness of the process reading from the queue.
- **Asynchronous Communication:** Message queues enable asynchronous processing, which allows you to put a message on the queue without processing it immediately.

http://blog.iron.io/2012/12/top-10-uses-for-message-queue.html?spref=tw

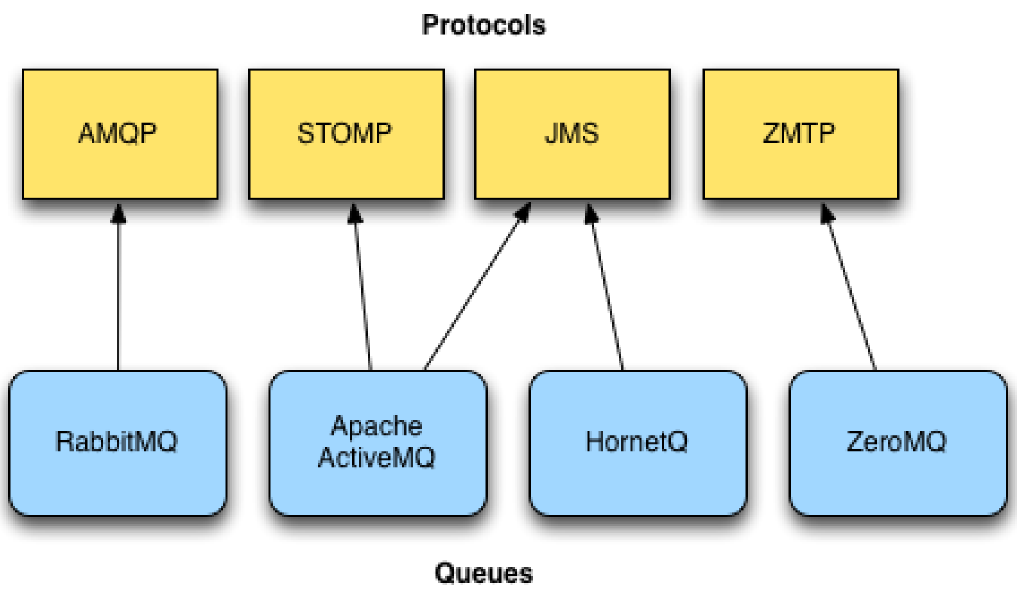
Using Message Manager

Disadvantages

- the need for explicit use of queues in distributed application;
- the complexity of implementation of **synchronous** exchange;
- some overhead when use the queue managers;
- the complexity of the response: reply may incur additional queues on each component that sends the messages.

Message Queue Services

- The MOM paradigm has had a long history in distributed applications.
- Message Queue Services (MQS) have been in use since the 1980's.
- The IBM WebSphere MQ is an example of such a facility.
- JMS – Java Message Service
- Microsoft Message Queuing (MSMQ - .NET)

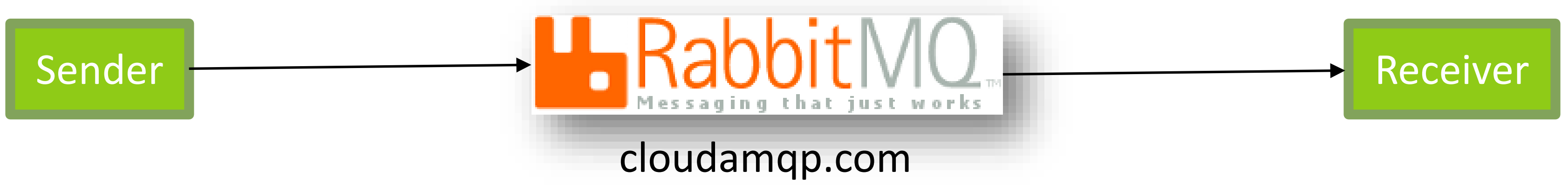


RabbitMQ vs ActiveMQ

- RabbitMQ (http://www.rabbitmq.com/)
  - Is based on the Erlang
  - Works on all major operating systems
  - Supports many platforms for developers (mostly - Python, PHP, Ruby)
  - Erlang-based configuration files
- Apache ActiveMQ (http://activemq.apache.org/)
  - Is based on the Java Message Service
  - Most often used in conjunction with Java-stack (Java, Scala, Clojure, etc).
  - Also supports STOMP (Ruby, PHP, Python).
  - XML-based configuration

Example of RabbitMQ usage

- RabbitMQ as a Service: http://www.cloudamqp.com/
- Sample Application: https://github.com/cloudamqp/java-amqp-example
- Start the Sender (OneOffProcess.java), after - 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 + "'");
    }
```

Serialization and data exchange formats

Marshaling

- **Marshalling** is a process of converting parameters to pass them via a network between processes withing remote call
- Marshaling
  - **by link** - an instance of the remote object resides on the server and does not leave it, and intermediaries are used for access
  - **by value -** the remote object is serialized and its copy is passed to another process

Data serialization formats

- Text:
  - **XML** - Extensible Markup Language
  - **JSON** - JavaScript Object Notation
  - **YAML** - YAML Ain't Markup Language (or Yet Another Markup Language)
- Binary
  - Protocol Buffers (Google)
  - MessagePack
  - Byte stream:
    - java.io.Serializable interface
    - .NET Serializable attribute

XML vs JSON

XML

```
<person>
  <firstName>John</firstName>
  <lastName>Smith</lastName>
  <address>
    <streetAddress>Downing Street 10
</streetAddress>
    <city>London</city>
    <postalCode>SW1A 2AA</postalCode>
  </address>
  <phoneNumbers>
    <phoneNumber>812 123-1234</phoneNumber>
    <phoneNumber>916 123-4567</phoneNumber>
  </phoneNumbers>
</person>
```

291 byte

JSON

```
{
   "firstName": "John",
   "lastName": "Smith",
   "address":  {
      "streetAddress": "Downing Street 10",
      "city": "London",
      "postalCode": "SW1A 2AA"
   },
   "phoneNumbers":  [
      "812 123-1234",
      "916 123-4567"
   ]
}
```

174 bytes

Google Protocol Buffers

- **Protocol Buffers** are a method of serializing structured data, proposed by Google in 2008, as an alternative to the XML (it was designed to be smaller and faster than XML).
- A software developer defines data structures (called *messages*) and services in a proto definition file (.proto) and compiles that with protoc.
- This compilation generates code that can be invoked by a sender or recipient of these data structures.
- Protocol Buffers are serialized into a compact, forwards-compatible, backwards-compatible **but not self-describing** binary wire format .
- PS: Used 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;
}
```

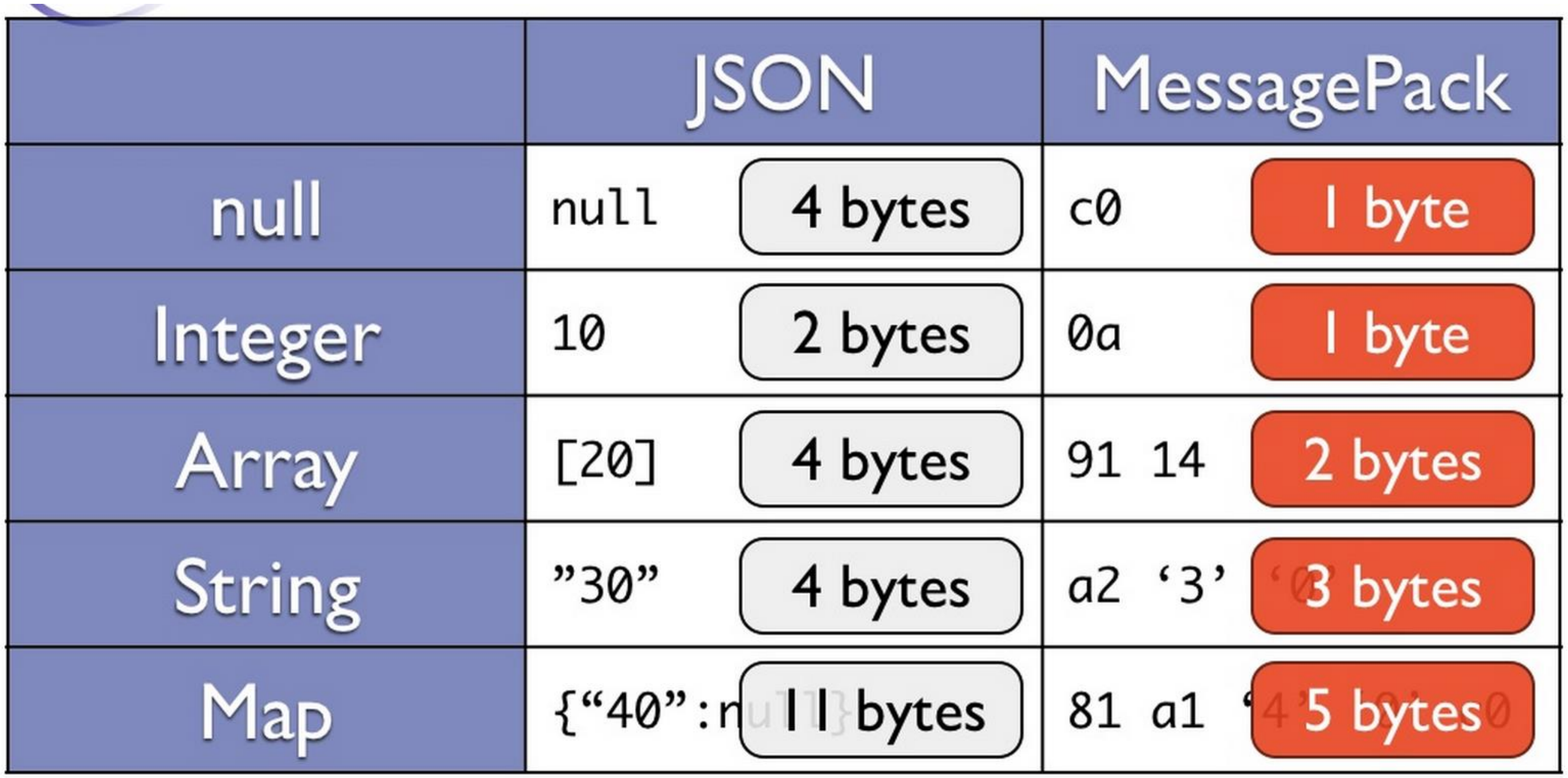
.proto

MessagePack

- MessagePack is an efficient binary serialization format. It lets you exchange data among multiple languages like JSON. But it's faster and smaller.
- Was designed to provide transparent conversion to JSON

Fixed-size data		Variable-sized data		
Type	Value	Type	Length	Body

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’ ‘0’	3 bytes
Map	{“40”:null}	11 bytes	81 a1 ‘4’ ‘0’	5 bytes

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

MessagePack

JSON

```
{
   "firstName": "John",
   "lastName": "Smith",
   "address":  {
      "streetAddress": "Downing Street
10",
      "city": "London",
      "postalCode": "SW1A 2AA"
   },
   "phoneNumbers":  [
      "812 123-1234",
      "916 123-4567"
   ]
}
```

174 bytes

```
84 a9 66 69 72 73 74 4e 61 6d 65 a4 4a 6f 68
6e a8 6c 61 73 74 4e 61 6d 65 a5 53 6d 69 74
68 a7 61 64 64 72 65 73 73 83 ad 73 74 72 65
65 74 41 64 64 72 65 73 73 b1 44 6f 77 6e 69
6e 67 20 53 74 72 65 65 74 20 31 30 a4 63 69
74 79 a6 4c 6f 6e 64 6f 6e aa 70 6f 73 74 61 6c
43 6f 64 65 a8 53 57 31 20 41 32 41 41 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)

144 bytes 83 %

http://msgpack.org/

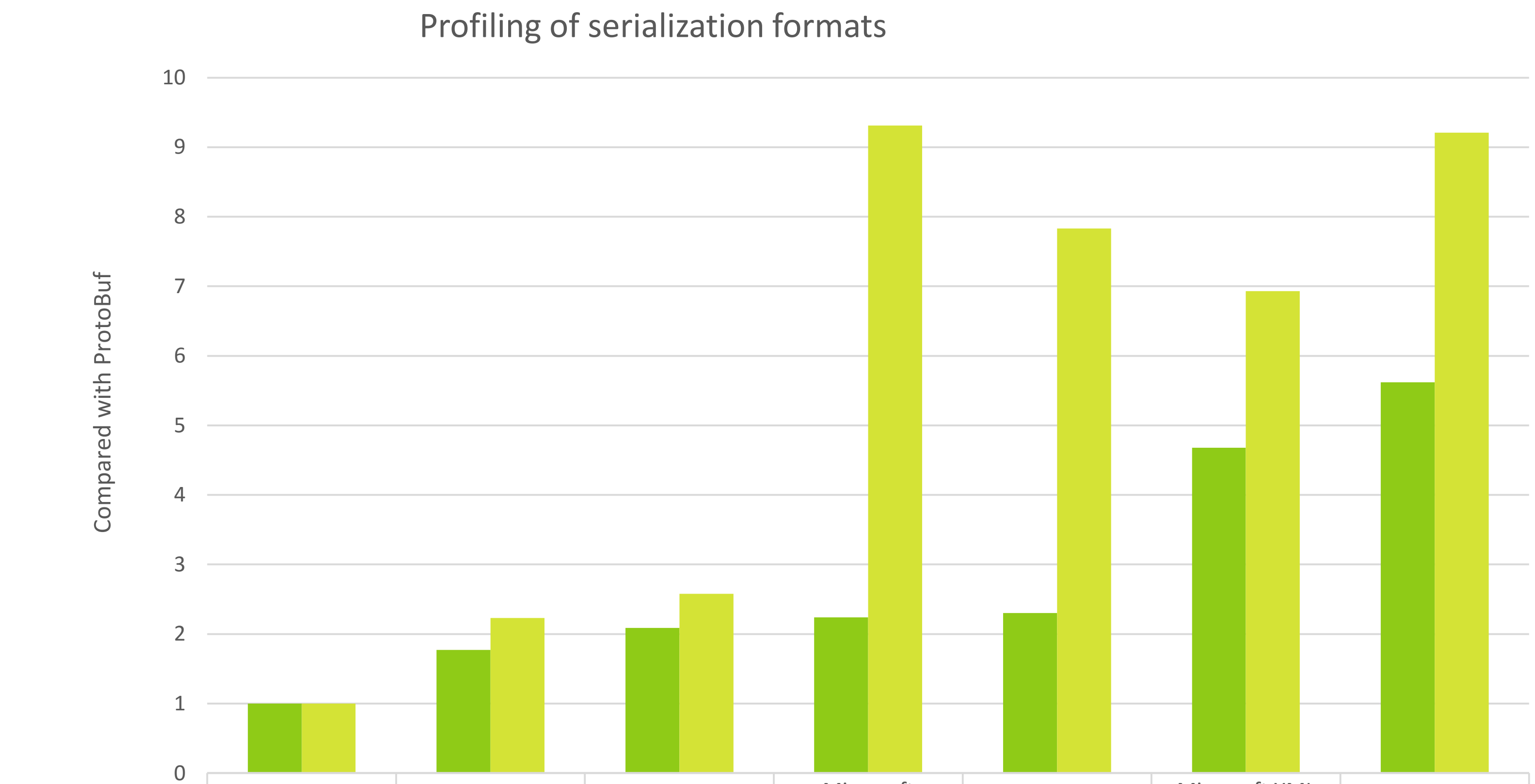
MessagePack

- JSON+ZIP VS MessagePack:
  - 50% drop in performance during packing / unpacking
  - Unable to work with data in stream mode (directly), you need all the data to provide decompression

Data serialization formats

- JSON
  - human readable/editable
  - can be parsed without knowing schema in advance
  - excellent browser support
  - less verbose than XML
- XML
  - human readable/editable
  - can be parsed without knowing schema in advance
  - standard for SOAP etc
  - good tooling support (XSD, XSLT, SAX, DOM, etc)
  - pretty verbose
- Protobuf (Google), MessagePack
  - very dense data (small output)
  - very fast processing
  - not intended for human eyes (dense binary)
- Protobuf (Google)
  - Built-in support of protocol versions (if you change the protocol, clients can work with the old version, while not updated)
  - hard to robustly decode without knowing the schema (data format is internally ambiguous, and needs schema to clarify)

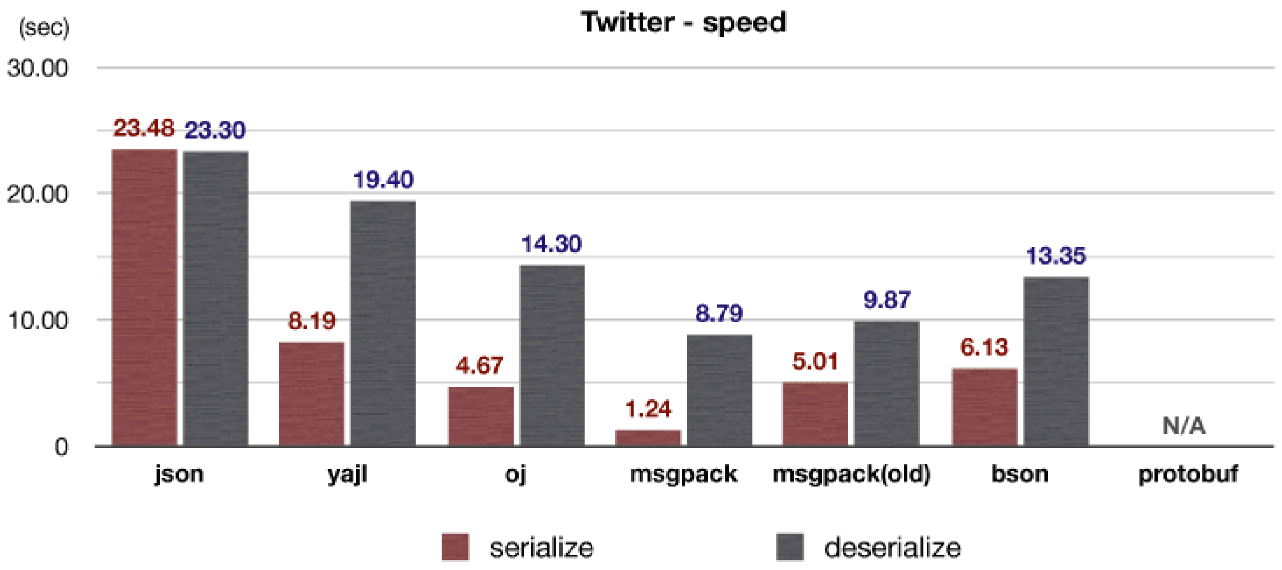
Data serialization formats

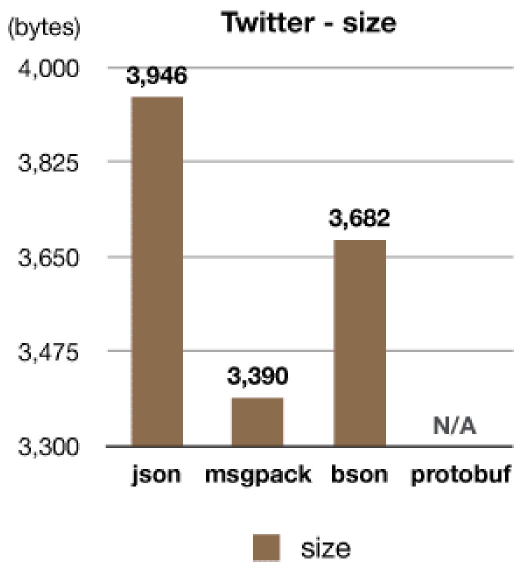


	ProtoBuf.net	ServiceStack TypeSerializer	ServiceStack JsonSerializer	Microsoft JsonDataContract Serializer	NewtonSoft.Json	Microsoft XML DataContractSerializer	Microsoft BinaryFormatter
🟩 Bigger (times)	1	1,77	2,09	2,24	2,3	4,68	5,62
🟩 Slower (times)	1	2,23	2,58	9,31	7,83	6,93	9,21

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

When to use which formats?

- **XML**
  - If the system provides a public API as an XML Web-service
  - Working with a "classical" system, which already uses XML as a standard data exchange
  - Required standard tools for verification and transformation (eg, to an HTML) (XSD, XSLT, SAX, DOM, etc.)
- **JSON**
  - If the system provides a public API as a REST-service
  - If clients are implemented in JavaScript
  - More compact than XML (2-2.5 times), and the easiest to read / edit – so wherever you want to use XML, think, may be worth using JSON.
- **MessagePack** – if you need high data processing speed and JSON compatibility, but you don't need the data to be readable/editable by the human.
- **Protobuf**
  - If data size and data processing time is essential
  - If you need to support protocol updates
  - If you implement an internal (non-public) protocol

Summary

- **A protocol** is a set of rules and agreements, describing the procedure for interaction between components of the system.
- There are options for direct communication in DCS and use message managers.
- RPC technology is used to call a function or procedure in a different address space
- RMI technology – development of RPC, provides transparent access to the methods of remote objects
- You need to choose a Data serialization format that fits your distributed system