External data representation

- https://developers.google.com/protocol-buffers/
- https://github.com/protobuf-c/protobuf-c
- http://www.drdobbs.com/web-development/after-xml-json-then-what/240151851
- http://www.digip.org/jansson/

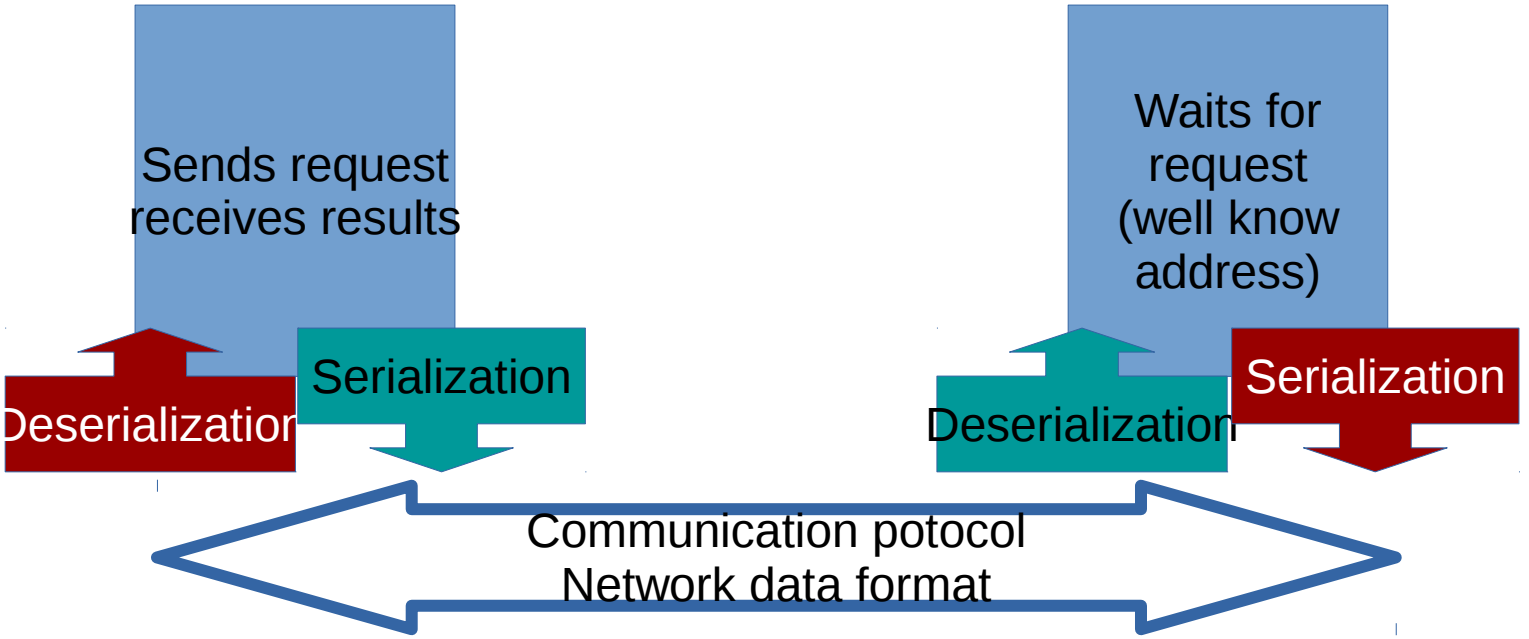
System data

- Internal data is represented as data structures
  - C structures/arrays
  - Java objects
- Transferred data is represented as byte sequences
- Data must be flatten to be transmitted
- Same data type can have multiple representations
  - e.g. floats/integers/characters

Data transmission

- Format of the transmitted data should be agreed
  - conversion to a common format
    - transmitter converts
    - receiver converts
  - transmitted in the sender format
    - receiver converts

Data transfer



- What kind of protocol to use, and what data to transmit?
- Efficient mechanism for storing and exchanging data

- Requirements
  - Correction
  - Efficiency
  - Interoperability (language/OS)
  - Ease to use

Data representation

- CORBA
  - Overdesigned and heavyweight
- Java object serialization
  - taylored to one environment: Java
- DCOM, COM+
  - taylored to one environment: Windows
- JSON, Plain Text, XML
  - Lack protocol description.
  - Programmer has to maintain both client and server code.
  - XML has high parsing overhead.
  - Relatively expensive to process; large due to repeated tags
- Binary

Binary - byteorder

- Byte oder on >16bits numbers depend on the processor architecture
- Can be done in two ways:
  - Big-endian:
    - lower addresses with higher order bits (ex: ARM *).
  - Little-endian:
    - lower addresses with lower order bits (ex: Intel x86).
- Integer 1000465 (0x000F4411),

Big-endian			Little-endian	
0x10003	11		0x10003	00
0x10002	44		0x10002	0F
0x10001	0F		0x10001	44
0x10000	00		0x10000	11

htons htonl
ntohs ntohl

- Big-endian advantages:
  - Integers are stored in the same order as strings (from left to right).
  - Number signal is on the “first byte” (base address) .
- Little-endian advantages:
  - Eases conversion between different length integers (ex: 12 is represented by 0x0C eor 0x000C).
- In the Internet,
  - Addresses are alwayes big-endian.
- The first ARPANET routers (named Interface Message Processor) were 16 bits Honeywell DDP-516 computers big-endian representation

htons htonl
ntohs ntohl

- uint32_t htonl(uint32_t hostlong);
  - The htonl() function converts the unsigned integer hostlong from host byte order tonetwork byte order.
- uint16_t htons(uint16_t hostshort);
  - The htons() function converts the unsigned short integer hostshort from host byte order to network byte order.
- uint32_t ntohl(uint32_t netlong);
  - The ntohl() function converts the unsigned integer netlong from network byte order to host byte order.
- uint16_t ntohs(uint16_t netshort);
  - The ntohs() function converts the unsigned short integer netshort from network byte order to host byte order.
- On the i386 the host byte order is Least Significant Byte first, whereas the network byte order, as used on the Internet, is Most Significant Byte first.

How to represent data?

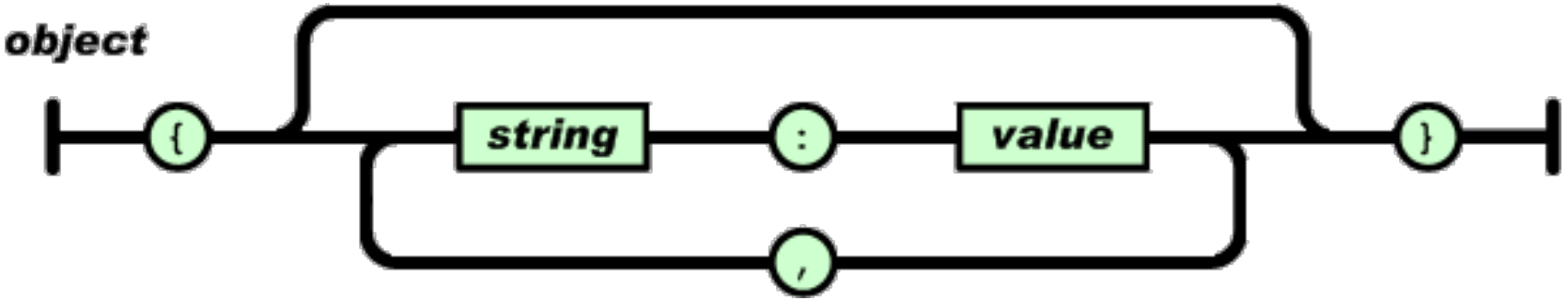
- How to represent data?
- Which data types do you want to support?
  - Base types, Flat types, Complex types
- How to encode data into the wire
- How to decode the data?
  - Self-describing (tags)
  - Implicit description (the ends know)
- Several answers:
  - Many frameworks do these things automatically

JSON

- JSON is built on two structures:
  - A collection of name/value pairs.
    - In various languages, this is realized as an object, record, struct, dictionary, hash table, keyed list, or associative array.
  - An ordered list of values.
    - In most languages, this is realized as an array, vector, list, or sequence.
- These are universal data structures.
  - Virtually all modern programming languages support them in one form or another.
  - It makes sense that a data format that is interchangeable with programming languages also be based on these structures.

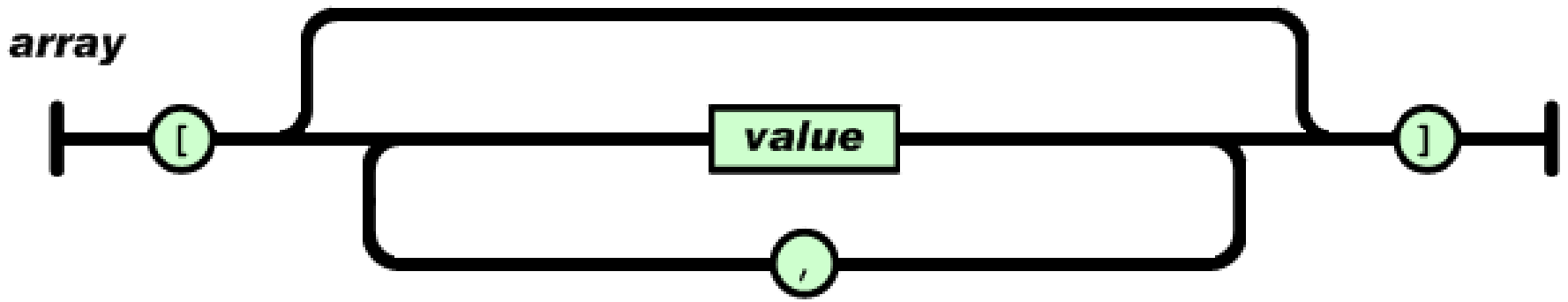
JSON

- An object is an unordered set of name/value pairs.
  - An object begins with { (left brace) and ends with } (right brace).
  - Each name is followed by : (colon) and the name/value pairs are separated by , (comma).



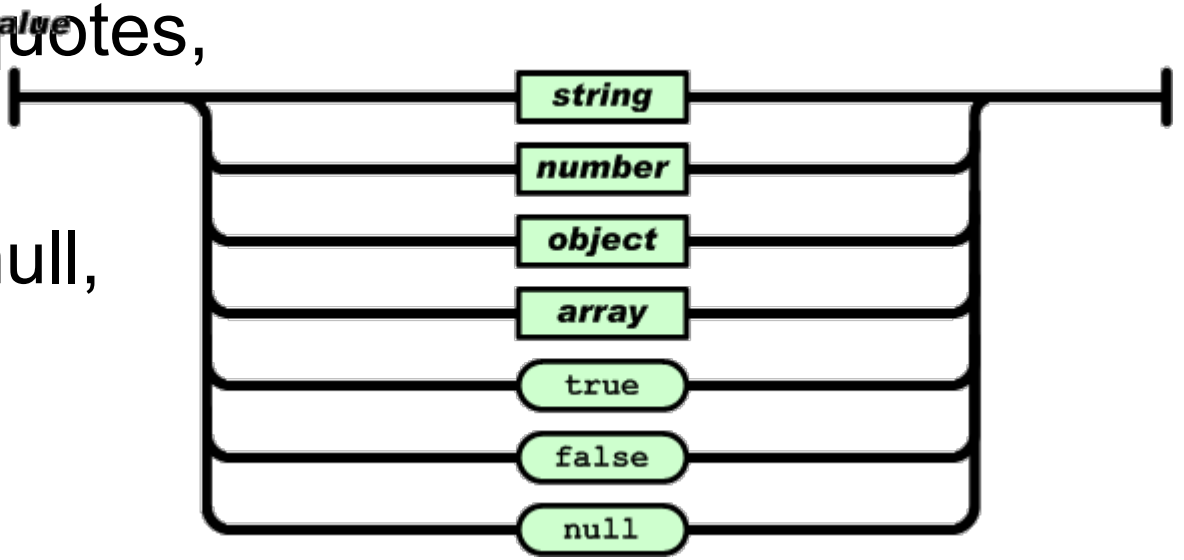
JSON

- An array is an ordered collection of values.
  - An array begins with [(left bracket) and ends with] (right bracket).
  - Values are separated by , (comma).



JSON

- A value can be
  - a string in double quotes,
  - or a number,
  - or true or false or null,
  - or an object
  - or an array.



- These structures can be nested.

http://www.digip.org/jansson/

```
Parse text:
root = json_loads(text, 0, &error);
if(!root){
    return 1;
}
```

```
Verify if is array
if(!json_is_array(root)){
    fprintf(stderr, "error: root is not an array\n");
    return 1;
}
```

```
Verify if is object
if(!json_is_object(data)){
        fprintf(stderr, "error: commit data is not an object\n");
        return 1;
    }
```

http://www.digip.org/jansson/

```
Get data from array
for(i = 0; i < json_array_size(root); i++){
    data = json_array_get(root, i);
….
Get data from object
sha = json_object_get(data, "sha");
if(!json_is_string(sha)){
        fprintf(stderr, "error: sha is not a string\n");
        return 1;
 }
message_text = json_string_value(sha);
```

http://www.digip.org/jansson/

- `json_t *json_array(void)`
- `int json_array_append(json_t *array, json_t *value)`
- `json_t *json_object(void)`
- `int json_object_set(json_t *object, const char *key, json_t *value`
- `json_t *json_integer(json_int_t value)`
- `char *json_dumps(const json_t *json, size_t flags)`

xml

- XML stands for eXtensible Markup Language
- XML is a markup language much like HTML
- XML was designed to store and transport data
- XML was designed to be self-descriptive
- XML is a W3C Recommendation

```
<note>
  <to>Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend!</body>
</note>
```

xml

- XML Documents Must Have a Root Element
- All XML Elements Must Have a Closing Tag
- XML Tags are Case Sensitive
- XML Elements Must be Properly Nested
- XML Attribute Values Must be Quoted
- White-space is Preserved in XML
- XML Stores New Line as LF
- Well Formed XML

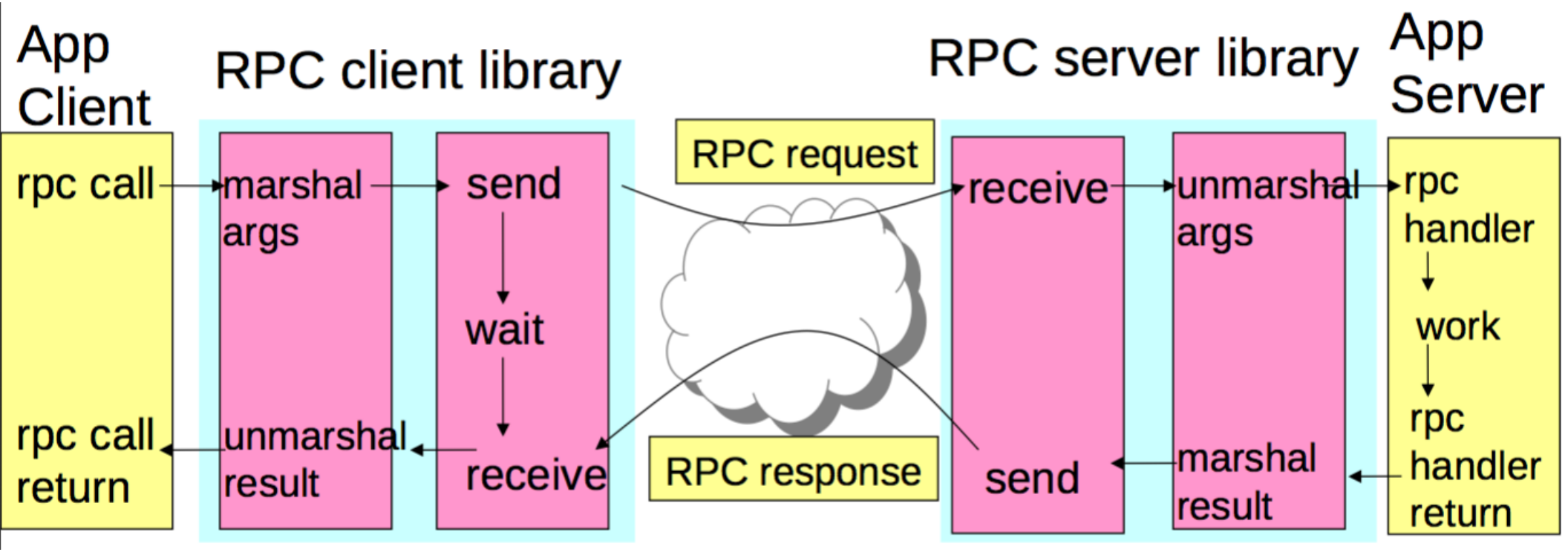
JSON vs XML

- Similarities:
  - Both are human readable
  - Both have very simple syntax
  - Both are hierarchical
  - Both are language independent
- Differences:
  - Syntax is different
  - JSON is less verbose
  - JSON includes arrays
  - Names in JSON must not be JavaScript reserved words
  - XML can be validated
  - JavaScript is not typically used on the server side
- Still require explicit parsing and processing by the programmer

Data Schema

- How to parse the encoded data?
- Two Extremes:
  - Self-describing data: tags
    - Additional information added to message to help in decoding
    - Examples: feld name, type, length
  - Implicit: the code at both ends “knows” how to decode the message
    - Interoperability depends on well defined protocol specification!
    - Very difficult to change

Frameworks


App Client
RPC client library
RPC server library
App Server
rpc call
marshal args
send
wait
receive
RPC request
RPC response
receive
unmarshal args
rpc handler
work
rpc handler return
send
marshal result
unmarshal result
rpc call return

Stub Generation

- Many systems generate stub code from independent specification: IDL
  - IDL – Interface Description Language
    - describes an interface in a language neutral way
- Separates logical description of data from
  - Dispatching code
  - Marshalling/unmarshalling code
  - Data wire format

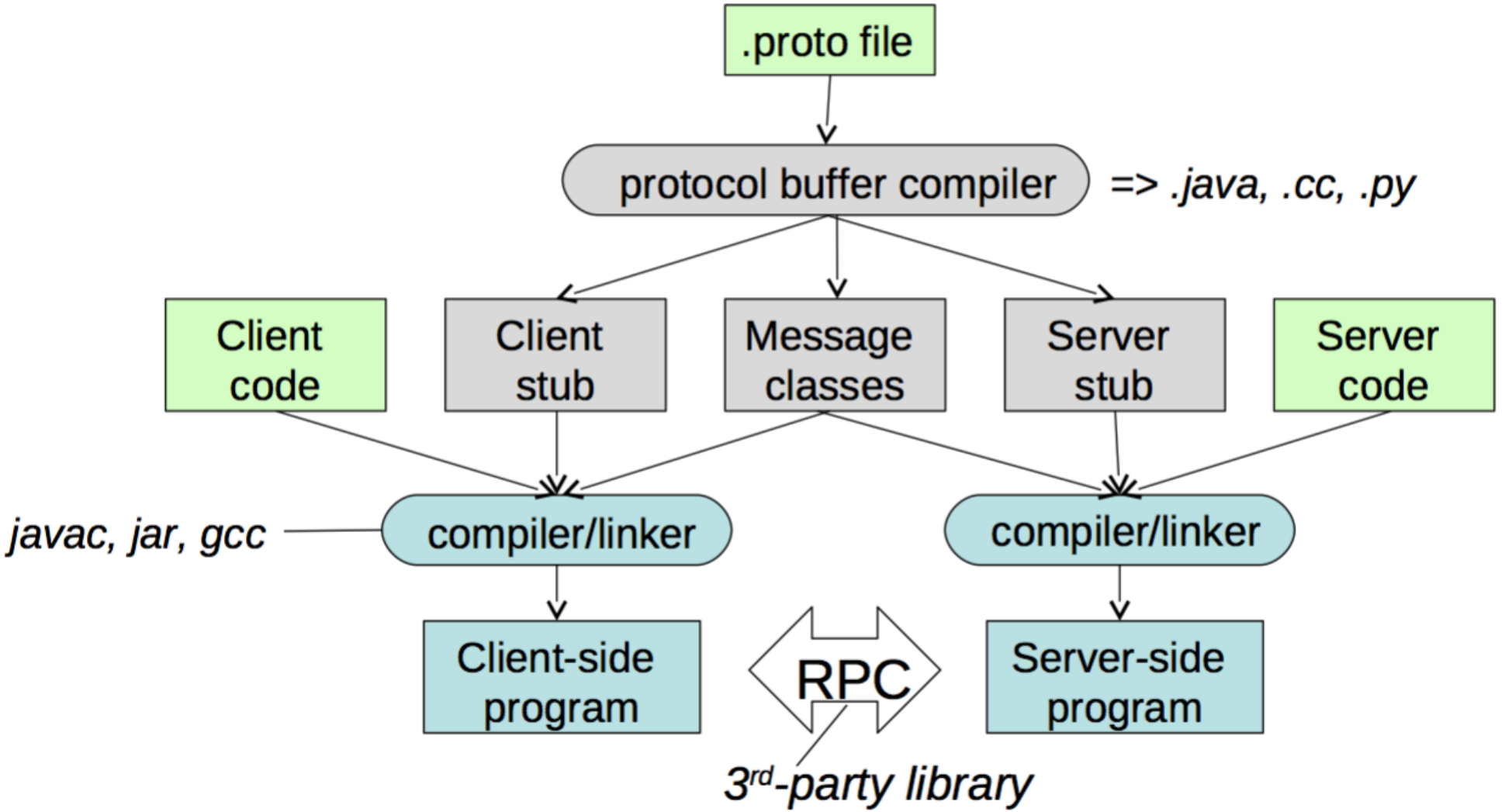
Google Protocol Buffers

- Defined by Google, released to the public
- Widely used internally and externally
- Supports common types, service definitions
- Natively generates C++/Java/Python code
  - Over 20 other supported by third parties
- Not a full RPC system, only does marshalling
  - Many third party RPC implementations
- Effcient binary encoding, readable text encoding
- Performance
  - 3 to 10 times smaller than XML
  - 20 to 100 times faster to process

Google Protocol Buffers

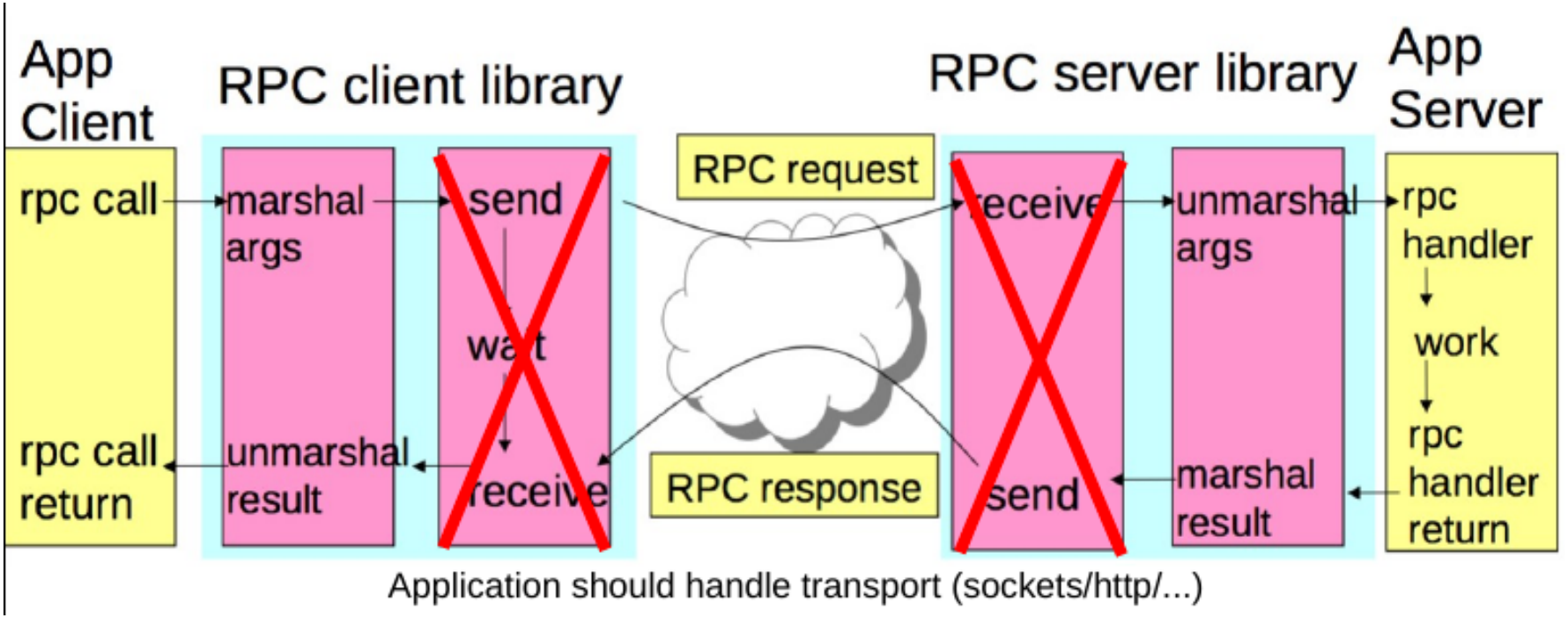
- Properties
  - Efficient, binary serialization
  - Support protocol evolution
    - Can add new parameters
    - Order in which I specify parameters is not important
    - Skip non-essential parameters
  - Supports types
    - which give you compile-time errors!
  - Supports somewhat complex structures
- Usage
  - Pattern: for each RPC call, define a new “message” type for its input and one for its output in a .proto file
  - Protocol buffers are used for other things, e.g., serializing data to non-relational databases
    - their backward compatible features make for nice long-term storage formats
  - Google uses them everywhere (50k proto buf definitions)

Google Protocol Buffers



Google Protocol Buffers

- Support service definitions and stub generation, but don't come with transport for RPC
  - There are third-party libraries for that



Goal of Protocol Buffer

- The goal of Protocol Buffer is to provide a language- and platform-neutral way to specify and serialize data such that:
  - Serialization process is efficient, extensible and simple to use
  - Serialized data can be stored or transmitted over the network
- In Protocol buffers, Google has designed a language to specify messages

Protocol Buffer Language

- Message contains uniquely numbered fields
- Field is represented by
  - field-type,
  - Data-type
  - Field-name
  - encoding-value
  - default value]
- Available data-types
  - Primitive data-type
  - int, float, bool, string, raw-bytes
  - Enumerated data-type
  - Nested Message
    - Allows structuring data into an hierarchy

```
message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }

  repeated PhoneNumber phone = 4;
}
```

Protocol Buffer Language

- Field-types can be:
  - Required fields
  - Optional fields
  - Repeated fields
    - Dynamically sized array
- Encoding-value
  - A unique number
    - =1 =2 …
  - represents a tag that a particular field has in the binary encoding of the message

```
message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;

  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }

  message PhoneNumber {
    required string number = 1;
    optional PhoneType type = 2 [default = HOME];
  }

  repeated PhoneNumber phone = 4;
}
```