Distributed Systems [Fall 2013]

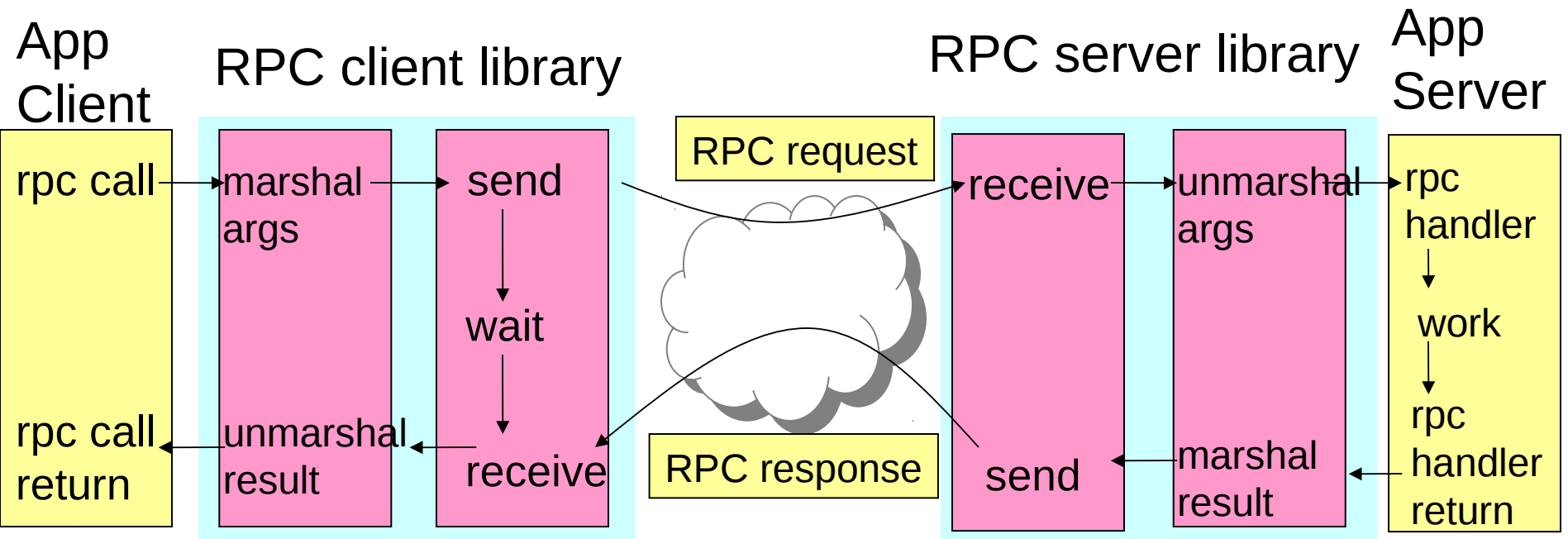
Lec 5: RPC Frameworks

Slide acks: Jinyang Li
(http://www.news.cs.nyu.edu/~jinyang/fa10/notes/ds-lec2.ppt)

Last Time (Reminder/Quiz)

- What's RPC? What are its goals?
- How does RPC work?
- What's data marshaling/unmarshalling (or serialization/de-serialization)?
- What are some challenges of RPC?
- What's at-most-once, at-least-once?

RPC Architecture




```
Client:
 { ...
    resp = foo("hello");
 }
```

```
Server:
int foo(char* arg) {
     …
}
```

Outline

- RPC challenges (from last lecture)
- RPC technologies
  - XML/RPC
  - Protocol buffers
  - Thrift
- Handouts!

Outline

- RPC challenges (from last lecture)
- RPC technologies
  - XML/RPC
  - Protocol buffers
  - Thrift
- Handouts!

RPC Technologies

- XML/RPC
  - Over HTTP, huge XML parsing overheads
- SOAP
  - Designed for web services via HTTP, huge XML overhead
- CORBA
  - Relatively comprehensive, but quite complex and heavy
- COM
  - Embraced mainly in Windows client software
- Protocol Buffers
  - Lightweight, developed by Google
- Thrift
  - Lightweight, supports services, developed by Facebook

XML/RPC

- Data serialization: XML
  - E.g.: RPC call to add(17, 13) results in this request:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<methodCall>
 <methodName>sample.add</methodName>
   <params>
      <param>
         <value><int>17</int></value>
      </param>
      <param>
         <value><int>13</int></value>
      </param>
   </params>
</methodCall>
```

- Data transmission protocol: HTTP

Example: Apache's XMLRPC Java Lib

- Handout: LISTING 1

- To remark:
  - How error-prone the untyped, vector-based param passing is
  - The verbosity of XML

The Problems with This Library

- XML is extremely verbose, which affects performance
- The library doesn't support protocol versioning
  - What happens if I want another param?
  - What happens if I reverse order of x and y?
    - In this case, nothing, but what if function weren't commutative?
  - What if I forget to add a param?
  - In general, lack of types makes it difficult to build & maintain code

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<methodCall>
 <methodName>sample.add</methodName>
   <params>
      <param>
         <value><int>17</int></value>
      </param>
      <param>
         <value><int>13</int></value>
      </param>
   </params>
</methodCall>
```

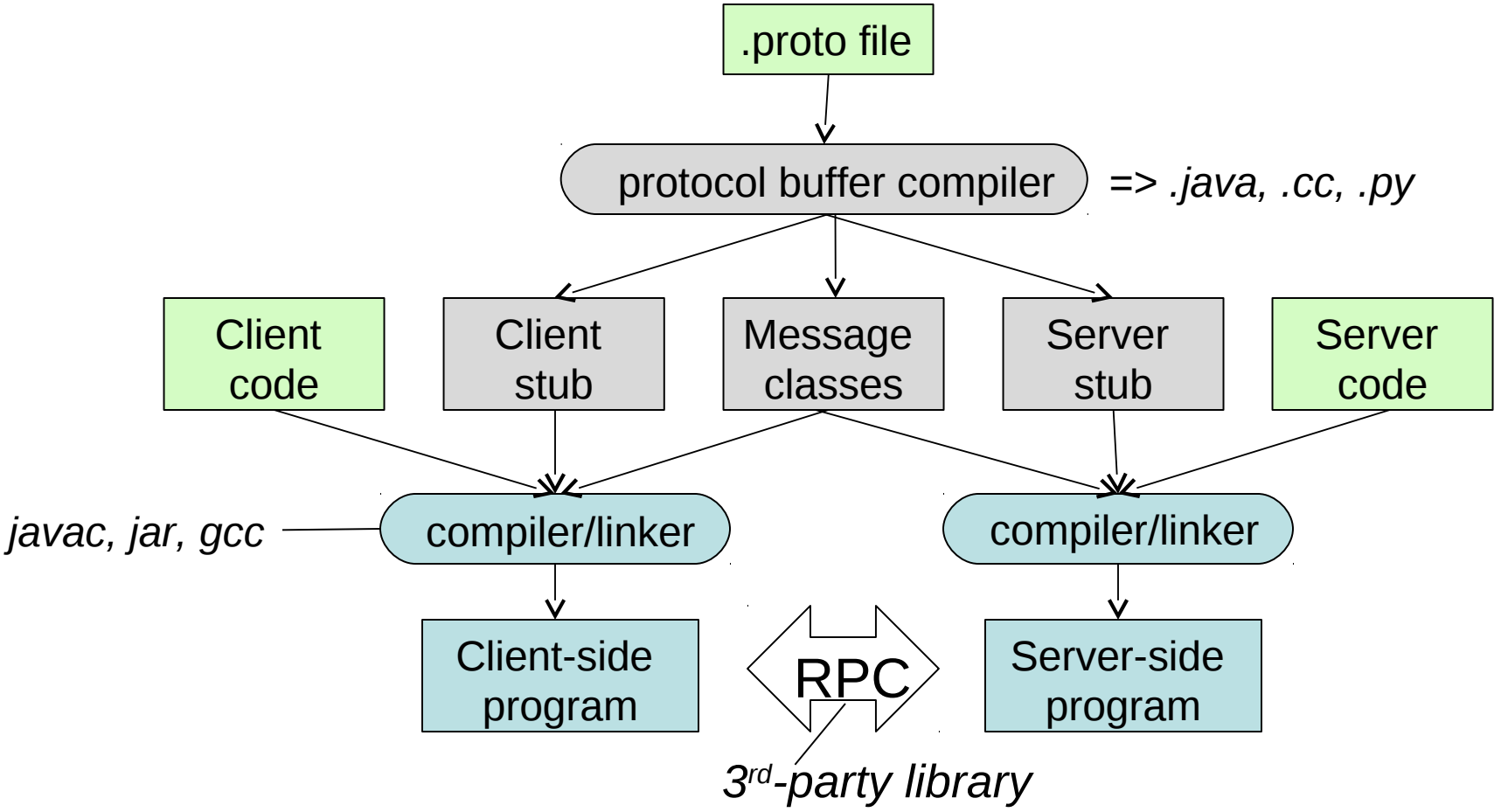
```
Vector params = new Vector();
params.addElement(new Integer(newParam));
params.addElement(new Integer(17));
params.addElement(new Integer(13));
```

- A more complex XML/RPC library could support types, this one just doesn't

Protocol Buffers

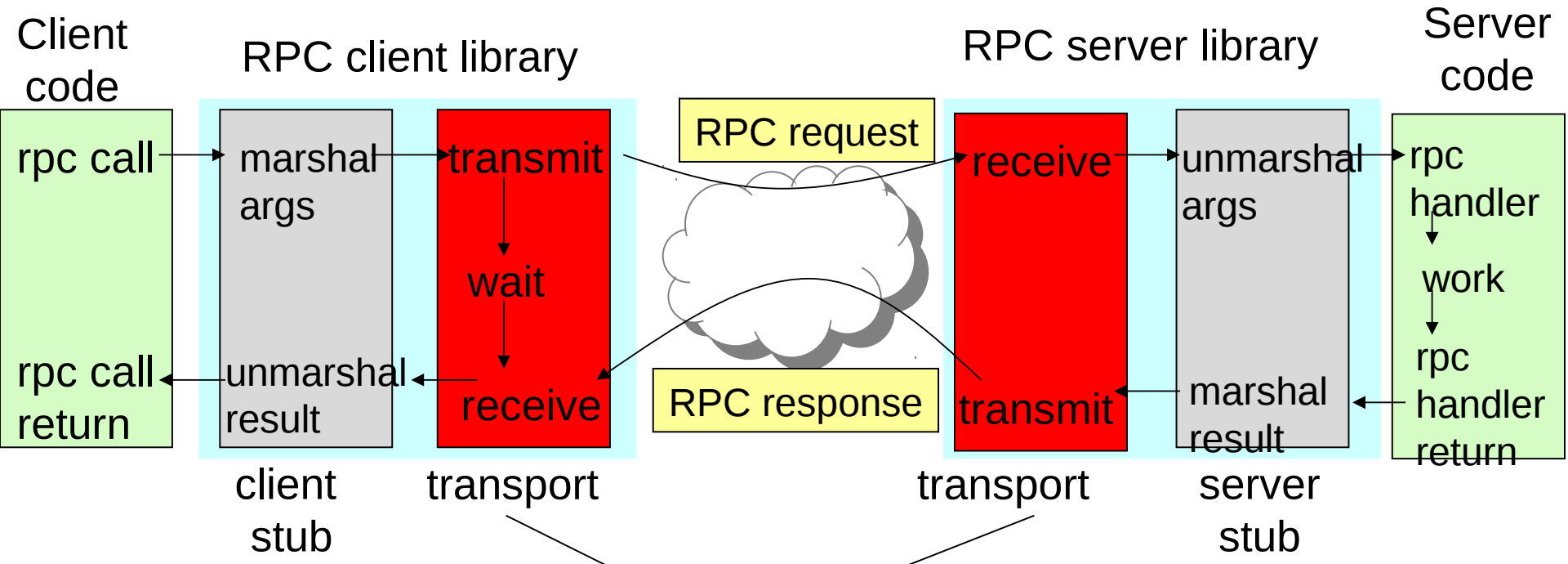
- Properties:
  - Efficient, binary serialization
  - Support protocol evolution
    - Can add new parameters
    - Order in which I specify parameters is not important
    - Skip non-essential parameters
  - Supports types, which give you compile-time errors!
  - Supports somewhat complex structures
- 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-compat features make for nice long-term storage formats
  - Google uses ‘em *everywhere* (48,162 proto buf definitions)

Protocol Buffer Workflow



Protocol Buffer Library Limitations

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



Example: Protocol Buffer Address Book

- Handout: LISTING 2

- To remark:
  - Field IDs, which allow protocol to evolve over time
    - IDs are sent along with values and uniquely identify the fields
    - IDs are written in stone – must never be changed in future
  - Some fields may be optional
    - You must never remove a non-optional field from a protobuf!
  - Repeated fields have no special ordering by default

(Note: just noticed a bug in handout: two person variables)

Versioning

- Without support for versioning, building distributed systems becomes a nightmare over time

```
if (version == 3) {
   ...
} else if (version > 4) {
   if (version == 5) {
     ...
  }
  ...
}
```

urgh!

- Protocol buffers, along with other solid RPC libraries, include support for versioning
- They make it hard for programmers to evolve their protocols in non-backward-compat ways

Example: Protocol Buffer Address Book

- Handout: LISTING 2

- To remark:
  - Field IDs, which allow protocol to evolve over time
    - IDs are sent along with values and uniquely identify the fields
    - IDs are written in stone – must never be changed in future
  - Some fields may be optional
    - You must never remove a non-optional field from a protobuf!
  - Repeated fields have no special ordering by default

Comparison: Protobuf vs. XML

- Protobufs are marshaled extremely efficiently
  - Binary format (as opposed to XML's textual format)
- Example (according to protobuf documentation):

XML

```
<person>
  <name>John Doe</name>
  <email>jdoe@example.com</email>
</person>
```

- size: 69 bytes (w/o whitespaces)
- parse: 5,000-10,000ns

Protobuf

```
person {
    1:"John Doe"
    3:"jdoe@example.com"
}
```

- size: 28 bytes
- parse: 100-200ns

- BUT: Do you see any problems, too?

https://developers.google.com/protocol-buffers/docs/overview

Thrift

- Similar in flavor to protocol buffer technology
- Advantages:
  - Supports somewhat more fancy types
    - Lists, sets, maps, exceptions, constants
  - Compiles to additional languages:
    - C#, Php, Ruby, Erlang, Haskell, …
  - Serializes to both binary and JSON
  - Incorporates RPC transport!
  - Supports streaming
    - I.e., server can start processing on parts of input!

Example: Thrift AddressBook

- Handout: LISTING 3

- To observe:
  - Very similar flavor to protocol buffers
  - Supports both ordered lists and unordered sets

RPC Summary

- RPC technology focuses on programming use and aims to:
  - Make distributed communication similar to local calls
  - Support protocol evolution
  - Make it hard to get it wrong
- Semantics are challenging
  - Can't really hide the network and make it all look local
- Performance is key
- You've learned about a few technologies, which you might use in future

Distributed Systems

Lecture 5: RPC Technologies

LISTING 1: Apache's XML/RPC Java Library Example

(http://www.tutorialspoint.com/xml-rpc/xml_rpc_examples.htm)

1.a) The client-side code:

```
import org.apache.xmlrpc.*;

public class JavaClient {
  public static void main(String [] args) {
    try {
      XmlRpcClient server = new XmlRpcClient("http://localhost/RPC2");

      Vector params = new Vector();
      params.addElement(new Integer(17));
      params.addElement(new Integer(13));

      Object result = server.execute("sample.add", params);
      int sum = ((Integer)result).intValue();
      System.out.println("The sum is: "+ sum);
    } catch (Exception exception) { // …
    }
  }
}
```

1.b) The server-side code:

```
import org.apache.xmlrpc.*;

public class RPCHandler {
  public Integer add(int x, int y) {
    return new Integer(x + y);
  }

  public static void main (String[] args) {
    try {
      WebServer server = new WebServer(80);
      server.addHandler("sample", new RPCHandler());  // register the handler class
      server.start();
    } catch (Exception exception) { // …
    }
  }
}
```

1.c) XML Marshaling:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<methodCall>
   <methodName>sample.add</methodName>
   <params>
       <param>
            <value><int>17</int></value>
       </param>
       <param>
             <value><int>13</int></value>
       </param>
   </params>
</methodCall>
```

LISTING 2: Protocol Buffer API Example

(https://developers.google.com/protocol-buffers/docs/overview)

2.a) Defining the protocol buffer:

You define a protocol buffer (or a set thereof) by writing their definitions in the protocol-buffer language into a **.proto** file. Example: **AddressBook.proto**, which defines protocol buffers for an address book:

```
package tutorial
option java_package = "com.example.tutorial";
option java_outer_classname = "AddressBookProtos";

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;
}

message AddressBook {
    repeated Person person = 1;
}
```

2.b) Compiling protocol buffers:

The protocol-buffer library provides a compiler, which takes in a **.proto** file and generates corresponding classes in a language of your choice, e.g. Java, Python, or C++ (third parties provide compiler extensions for other languages, too).

```
# $PROTOC_HOME/bin/protoc –java_out $PWD AddressBook.proto
        // generates com.example.tutorial.AddressBookProtos.java, with two classes:
        // Person and AddressBook
```

2.c) Using protobufs:

We can serialize and de-serialize protocol buffer structures to/from input and output streams. These streams can be backed either by some network channel or by files or even by database connections.

```
import com.example.tutorial.AddressBookProtos.Person;
import com.example.tutorial.AddressBookProtos.AddressBook;

public class HandleAddressBook {
  public static void createAndSerializeAddressBook(OutputStream output) {
    Person.Builder person = Person.newBuilder();
    person.setId(1234);
    person.setName(“John Doe”);

    Person.PhoneNumber.Builder phoneNumber =
        Person.PhoneNumber.newBuilder().setNumber(“102-203-4005”);
    phoneNumber.setType(Person.PhoneType.MOBILE);
    person.addPhone(phoneNumber);
    // Can add other phone numbers.
```

```
    // person.setEmail("johndoe@email.com");  // this is optional –may or may not add it.

    Person person = person.build();  // generate the Person object.
    AddressBook.Builder addressBook = AddressBook.newBuilder();
    addressBook.addPerson(person);
    // Add other persons.

    // Write the new address book to an OutputStream (can be backed by a file, a
    // socket stream, etc.).
    addressBook.build().writeTo(output);
  }

  public static void readAndDisplayAddressBook(InputStream input) {
    AddressBook addressBook = AddressBook.parseFrom(input);
    for (Person person: addressBook.getPersonList()) {

      System.out.println("Person ID: " + person.getId());
      System.out.println("  Name: " + person.getName());
      if (person.hasEmail()) {
        System.out.println("  E-mail address: " + person.getEmail());
      }

      for (Person.PhoneNumber phoneNumber : person.getPhoneList()) {
        switch (phoneNumber.getType()) {
        case MOBILE:
          System.out.print("  Mobile phone #: "); break;
        case HOME:
          System.out.print("  Home phone #: "); break;
        case WORK:
          System.out.print("  Work phone #: "); break;
        }
        System.out.println(phoneNumber.getNumber());
      }
    }
  }
}
```

2.d) Services

Protocol buffers let us define **services**, which describe RPC functions exported by a server and used by clients. The **protoc** compiler will generate stubs for these RPC functions. For example, you can include the following definitions in the `AddressBook.proto` file, as well:

```
service AddressBookService {
  rpc searchForPerson(SearchRequest) returns (Person);  // might want to wrap the
      // returned Person into a new response type, which also includes error signaling.
  ...
}
message SearchRequest {
  string query = 1;  // e.g., can be the name, the phone number, etc…
}
```

The protoc compiler will then generate a Stub class for the service (AddressBookService.Stub), which will contain all of its functions (searchForPerson, ...). Please see https://developers.google.com/protocol-buffers/docs/proto#services for details on how to use services.

LISTING 3: Thrift API Example

(http://www.scribd.com/doc/95866167/Thrift-Protobuf)

3.a) Thrift structures:

Language is somewhat different, but flavor is the same: you create a **.thrift** file, compile it, and link the resulting code with your own. Example of **`AddressBook.thrift`**:

```
namespace java tutorial
namespace csharp Tutorial

enum PhoneType {
    MOBILE = 1,
    HOME = 2,
    WORK = 3
}

struct PhoneNumber {
    1: string number,
    2: PhoneType type = 2
}

struct Person {
    1: string name,
    2: i32 id,
    3: string email,
    4: set<PhoneNumber> phone
}

struct AddressBook {
    1: list<Person> person
}
```

3.b) Thrift API:

```
# $THRIFT_ROOT/bin/thrift -gen-java tutorial.thrift
    // code will be generated in gen-java/*.java

// Create new object and populate its fields.
AddressBook addressBook = new AddressBook(name, id, ...)

// Serialize:
TSerializer serializer = new TSerializer(newTBinaryProtocol.Factory());
byte[] bytes = serializer.serialize(addressBook);
// Send the bytes over some stream.

// De-serialize:
TDeserializer deserializer = new TDeserializer(new TBinaryProtocol.Factory());
deserializer.deserialize(addressBook, bytes);
// Do something with addressBook.
```

DISCLAIMER

All code listed in this document is approximate, does not do error handling, and in some cases may not even compile. Use it to get a sense for what these technologies are about, but refer to docs for in-depth guidance.