Serialisation: a Comparison

Jeff Abrahamson

Jellybooks

In pace requiescat

Corba

Basics

- Backwards compatible
- Forwards compatible
- Efficiency (space, time)

Subtleties

- Community
- Safety
- Accent

Frameworks considered

Good:

- Thrift (Facebook, now Apache)
- Protobuf (Google)

Not so good:

- Boost Serialisation
- Json

Don’t Even

- CORBA (1991) (cf. Vasa, 1628)
- XML-RPC (1998)
- SOAP (1998, successor to XML-RPC)

RPC

- Protobuf (no)
- Thrift (yes)
- Boost (no)
- Json (no)
- Cap’n Proto (yes)

RPC

Some popular RPC communication libraries:

- ZMQ
- RabbitMQ
- Etch (?)
- ZeroC ICE (?)
- Apache Qpid and AMQP (?)

But this isn't a talk about RPC.

Protobuf and Thrift

- IDL
- Proven scalable
- Community
- Similar efficiency

Protobuf

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

  enum PhoneType {
    MOBILE = 0;  // ...
  }

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

  repeated PhoneNumber phone = 4;
}
```

Protobuf

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

Protobuf

```
// name
inline bool has_name() const;
inline void clear_name();
inline const ::std::string& name() const;
inline void set_name(const ::std::string& value);
inline void set_name(const char* value);
inline ::std::string* mutable_name();
```

Protobuf

```
//Checks if all the required fields have been set.
bool IsInitialized() const;

// Returns a human-readable representation of the
// message, particularly useful for debugging.
string DebugString() const;

// Overwrites the message with the given message's
// values.
void CopyFrom(const Person& from);

// Clears all the elements back to the empty state.
void Clear();
```

Protobuf

```
// Serializes the message and stores the bytes in
// the given string. Note that the bytes are binary
// not text; we only use the string class as a
// convenient container.
bool SerializeToString(string* output) const;

// Parses a message from the given string.
bool ParseFromString(const string& data);

// Writes the message to the given C++ ostream.
bool SerializeToOstream(ostream* output) const;

// Parses a message from the given C++ istream.
bool ParseFromIstream(istream* input);
```

Protobuf

```
AddressBook address_book;

Person* person = address_book.add_person();
person->set_id(id);
person->set_name(name);
if (!email.empty()) {
    person->set_email(email);
}

Person::PhoneNumber* phone_number
    = person->add_phone();
phone_number->set_number(number);
phone_number->set_type(Person::MOBILE);
```

Protobuf

```
for (int j = 0; j < person.phone_size(); j++) {
    const Person::PhoneNumber&
            phone_number = person.phone(j);
```

Thrift

```
const map<string,string> MAPCONSTANT =
    {'hello':'world',
     'goodnight':'moon'}

enum Operation {
  ADD = 1,
  // ...
}

struct Work {
  1: i32 num1 = 0,
  2: i32 num2,
  3: Operation op,
  4: optional string comment,
}
```

Thrift

```
exception InvalidOperation {
  1: i32 whatOp,
  2: string why
}

struct SharedStruct {
  1: i32 key
  2: string value
}

service SharedService {
  SharedStruct getStruct(1: i32 key)
}
```

Thrift

```
service Calculator extends shared.SharedService {
    void ping(),
    i32 add(1:i32 num1, 2:i32 num2),
    i32 calculate(1:i32 logid, 2:Work w)
            throws (1:InvalidOperation ouch),

    /**
     * This method has a oneway modifier. That
     * means the client only makes a request and
     * does not listen for any response at
     * all. Oneway methods must be void.
     */
    oneway void zip()
}
```

Thrift

```
boost::shared_ptr<TTransport> socket(
    new TSocket("localhost", 9090));
boost::shared_ptr<TTransport> transport(
    new TBufferedTransport(socket));
boost::shared_ptr<TProtocol> protocol(
    new TBinaryProtocol(transport));
CalculatorClient client(protocol);
```

Thrift

```
transport->open();

client.ping();
cout << "1 + 1 = " << client.add(1, 1) << endl;
```

Thrift

```
Work work;
work.op = Operation::DIVIDE;
work.num1 = 1;
work.num2 = 0;

try {
    client.calculate(1, work);
    cout << "Division by zero!" << endl;
} catch (InvalidOperation& io) {
    cout << "InvalidOperation: " << io.why << endl;
    cout << io << endl;  // Same thing.
}
```

Thrift

```
SharedStruct ss;
client.getStruct(ss, 1);
cout << "Received log: " << ss << endl;

transport->close();
```

Thrift

```
int main() {
  boost::shared_ptr<TProtocolFactory>
          protocolFactory(
              new TBinaryProtocolFactory());
  boost::shared_ptr<CalculatorHandler>
          handler(new CalculatorHandler());
  boost::shared_ptr<TProcessor> processor(
      new CalculatorProcessor(handler));
  boost::shared_ptr<TServerTransport>
          serverTransport(new TServerSocket(9090));
  boost::shared_ptr<TTransportFactory>
          transportFactory(
              new TBufferedTransportFactory());

  TSimpleServer server(processor, serverTransport,
                       transportFactory,
                       protocolFactory);
```

Thrift

```
// Server

int32_t add(const int32_t n1,
            const int32_t n2) { ...

int32_t calculate(const int32_t logid,
                  const Work& work) { ...

void getStruct(SharedStruct& ret,
               const int32_t logid) { ...

void zip() { ...
```

Thrift

```
boost::shared_ptr<TFileTransport>
    transport(new TFileTransport(filename));
boost::shared_ptr<TBinaryProtocol>
    protocol(new TBinaryProtocol(transport));
myObj.write(protocol.get());
```

Boost

```
// Save: << or the & operator
ar << data;
ar & data;

// Load: >> or the & operator.
ar >> data;
ar & data;
```

Boost

```
// include headers that implement an archive in
// simple text format
#include <boost/archive/text_oarchive.hpp>
#include <boost/archive/text_iarchive.hpp>
```

Boost

```
class gps_position {
  private:
    friend class boost::serialization::access;
    // When the class Archive corresponds to an
    // output archive, the & operator is defined
    // similar to <<.  Likewise, when the class
    // Archive is a type of input archive the &
    // operator is defined similar to >>.
    template<class Archive>
    void serialize(Archive & ar,
                   const unsigned int version) {
        ar & degrees;
        ar & minutes;
        ar & seconds;
    }
    int degrees, minutes;
    float seconds;
```

Boost

```
int main() {
    std::ofstream ofs("filename");
    const gps_position g(35, 59, 24.567f);

    {
        boost::archive::text_oarchive oa(ofs);
        oa << g;
    }

    gps_position newg;  // Later...
    {
        std::ifstream ifs("filename");
        boost::archive::text_iarchive ia(ifs);
        ia >> newg;
    }
    return 0;
}
```

Boost

```
// Non-intrusive version.
template<class Archive>
void serialize(Archive & ar, gps_position & g,
               const unsigned int version) {
    ar & g.degrees;
    ar & g.minutes;
    ar & g.seconds;
}
```

Boost

```
// Serializable members.
class bus_stop {
    friend class boost::serialization::access;
    template<class Archive>
    void serialize(Archive & ar,
                   const unsigned int version) {
        ar & latitude;
        ar & longitude;
    }
    gps_position latitude;
    gps_position longitude;
```

Boost

```
// Derived classes.
#include <boost/serialization/base_object.hpp>

class bus_stop_corner : public bus_stop {
    friend class boost::serialization::access;
    template<class Archive>
    void serialize(Archive & ar,
                   const unsigned int version) {
        // serialize base class information
        ar & boost::serialization::base_object<
            bus_stop>(*this);
        ar & street1;
        ar & street2;
    }
    std::string street1;
    std::string street2;
```

Boost

```
// Pointers.
class bus_route {
    friend class boost::serialization::access;
    bus_stop* stops[10];
    template<class Archive>
    void serialize(Archive & ar,
                   const unsigned int version)
{
        for(int i = 0; i < 10; ++i) {
            ar & stops[i];
        }
    }
```

Boost

```
// Arrays.
class bus_route {
    friend class boost::serialization::access;
    bus_stop* stops[10];
    template<class Archive>
    void serialize(Archive & ar,
                   const unsigned int version) {
        ar & stops;
    }
```

Boost

```
// STL.
#include <boost/serialization/list.hpp>

class bus_route {
    friend class boost::serialization::access;
    std::list<bus_stop*> stops;
    template<class Archive>
    void serialize(Archive & ar,
                   const unsigned int version) {
        ar & stops;
    }
```

Boost

```
// Versioning (1).
#include <boost/serialization/list.hpp>
#include <boost/serialization/string.hpp>

class bus_route {
    friend class boost::serialization::access;
    std::list<bus_stop*> stops;
    std::string driver_name;
    template<class Archive>
    void serialize(Archive & ar,
                   const unsigned int version) {
        ar & driver_name;  // !!
        ar & stops;
    }
```

Boost

```
// Versioning (2).
#include <boost/serialization/list.hpp>
#include <boost/serialization/string.hpp>
#include <boost/serialization/version.hpp>

class bus_route {
    friend class boost::serialization::access;
    std::list<bus_stop *> stops;
    std::string driver_name;
    template<class Archive>
    void serialize(Archive & ar,
                   const unsigned int version) {
        if(version > 0)
            ar & driver_name;
        ar & stops;
    }
```

Boost

```
BOOST_CLASS_VERSION(bus_route, 1)
```

Json

- Ubiquitous
- Human readable
- No validation or error checking
- Verbose

Cf. Bson, MessagePack, ...

Questions?

https://github.com/JeffAbrahamson/talks/

jeff@purple.com