Data Serialization

Paulo Sérgio Almeida

Grupo de Sistemas Distribuídos
Departamento de Informática
Universidade do Minho

Kinds of heterogeneity

Heterogeneity in distributed systems may exist at several levels:

- Hardware;
- Programming Languages;
- Middleware.

Hardware heterogeneity

Many reasons to mix different hardware:

- application availability for the OS/Hardware combination;
- gradual machine replacement over time;

Different hardware may mean:

- different data representation: need for neutral representation;
- different machine code: obstacle to code migration;
  - implementation for each machine;
  - use of virtual machines; compile once and run everywhere.

Language heterogeneity

Many reasons to use/mix different languages:

- language suitability to problem;
- availability of libraries;
- legacy applications;

Tools to cope with language heterogeneity:

- data serialization formats + libraries;
- language neutral IDL + language mappings;
- code generation by IDL compiler; e.g. *stubs*, *skeletons*.

Middleware heterogeneity

Motivation:

- middleware *A* is popular; later middleware *B* is preferred;
- or, middleware *A* and *B* are suitable to different scenarios;
- two applications, built using *A* and *B*, need to interoperate;

Multiple vendors of single middleware:

- different vendor implementations should interoperate;
- does not happen without a wire-protocol standard;
- e.g., CORBA before IIOP, with only language-level standards;

Different middleware interoperability:

- interoperability protocols through *middleware bridges*;
- difficult for undefined/opaque/proprietary wire protocols;

Better to use simple middleware with defined wire-protocols.

Data serialization and remote invocation protocols

- Data serialization formats allow data structures to be sent over the network; e.g. ONC XDR, CORBA CDR, JSON.
- Remote invocation protocols, defined over transport protocols such as TCP, are also defined; e.g., ONC RPC, CORBA IIOP, JSON-RPC.
- Commonly described over an underlying bidirectional *stream*;
- Ideally, we should be able to choose each layer independently;
- Many RPC frameworks bundle them together;

Data serialization features

- Language specific versus language agnostic;
  e.g., Kryo versus JSON, CDR, Avro, Thrift, Protocol Buffers;
- Binary versus text-based (human readable);
  e.g., Kryo, CDR, Thrift, Protocol Buffers versus JSON;
- Does it allow sub-structure sharing / references?
  e.g., Kryo, CDR, YAML versus JSON, Avro, Thrift, Protocol Buffers;
- Versioning and Extensibility;
  does it allow different versions to co-exist?
- Self-description;
  how much data describes itself / needs external information;
  e.g. no info (e.g, XDR, CDR), field+type info (e.g., JSON);
- Encoding/decoding step vs neutral in-memory representation (binary formats);

Versioning and Extensibility

- Versioning: to allow backward and forward compatibility along a sequence of versions;
  - adding field
  - renaming field
  - removing field
  - changing field type
- Distributed extensibility: to allow extensions concurrently developed to interoperate;
  - the same concerns as for versioning and
  - possible name collisions in new field
  - sometimes partially addressed with namespaces, which themselves introduce complexity;
  - complex problem; not our focus;

Self-description

Where is the information that allows data to be decoded?

- Receiver code;
- External schema;
- Data stream itself;

Possible information in the data stream:

- Names of fields;
- Type information;

Self-description: some possibilities (1)

- No description + code:
  - Sender and receiver (code) know exact structure;
  - Manual coding or IDL + code generation;
  - e.g., XDR, CORBA IIOP;
- No description + separate schema:
  - separate schema is used to decode the data;
  - e.g., Avro;
- Field tagging + type information + code;
  - sender and receiver (code) know expected structure;
  - extract relevant fields, skip unknown fields, default missing;
  - e.g., Thrift, Protocol Buffers;

Self-description: some possibilities (2)

- Type and field names + separate schema for type information;
  - external schema contains type information to validate and decode;
  - e.g., some uses of XML;
- Field names + type information;
  - self-description of both field and type information;
  - e.g., JSON;

Binary formats

- Examples: XDR, CORBA CDR, Thrift, Protocol Buffers;
- Compact and allow a computationally efficient serialization;
- May use standards (e.g., IEEE 754) for floatint point;
- Have many design options, e.g.:
  - Alignment in serialization stream;
  - Endianness (little/big endian) treatment;
  - Fixed vs variable size fields;

Binary formats options/features (1)

- Alignment in serialization stream:
  - with alignment, resulting in *padding*; e.g., CDR;
  - without padding between fields; more compact; most formats;
- Endianness (little/big endian) treatment:
  - fixed neutral format; e.g., *network order*; most formats;
  - anotating format at sender; receiver converts only if different (*receiver makes it right*); more efficient; e.g. CDR.
- Fixed vs variable size fields:
  - For integer fields or describing the size of strings;
  - Most strings are small; most integers are small;
  - Fixed size assumes worst case;
    e.g., 4 byte string length, 8 byte integers in Thrift BinaryProtocol;
  - Variable size may use 1 or 2 bytes in the more common case; e.g., Protocol Buffers;

Binary formats options/features (2)

- Outer object before vs encompassing inner objects:
  - possibility of incremental reads and processing partial message;
- Random access:
  - possibility of acessing one field without parsing the whole message
- Zero-copy:
  - possibility of accessing message in-place;
- Whether default fields take space on message;
- Whether pointers take space on message;
- Whether (de-serialized) message is usable as mutable state;
- Arena allocation vs individual allocation of each object;

Neutral in-memory representation

- Most schemes involve an encoding/decoding step;
- Some binary formats use a neutral in-memory representation:
  - Cap'n Proto
  - Simple Binary Encoding
  - FlatBuffers
- Data is kept in memory already "serialized";
  - neutral format, defining endianess, aligment, etc;
  - position independent through offset based pointers;
- Designed for random access and incremental access;
- Allows immediate sending or reading (e.g. mmapped file);
- Message usable only as immutable state;
- Allows inter-process communication by sharing memory;
  - even across different languages;

Text based formats

- Convert binary values (e.g., numbers) into a textual representation;
- Less compact than binary formats;
- Much slower than binary formats (10 to 100 times);
  - convertion between binary and decimal number represention;
  - *parsing*;
- Increased popularity after WWW appeared;
- XML, popular with structured documents, was overused everywhere, including for data serialization;
- Other formats were proposed specifically for data serialization; e.g, YAML, JSON;
- JSON (JavaScript Object Notation) is becoming the more important text-based data serialization format;

Problems with XML for data serialization

XML is appropriate in the *document centric* use, but very inappropriate in the *data centric* use:

- It does not support natively essential data structures, like lists/sequences/arrays and maps/dictionaries/hashes;
- Mappings from/to language data structures may be:
  - complex;
  - arbitrary (with no single natural normal form);
  - possibly incompatible between themselves;
- XML processing is complex, e.g.:
  - *callbacks* (SAX);
  - tree manipulation (DOM).
- Serialization is costly in size and very costly in processing time;

Problems with XML for data serialization: arrays

- Lists/sequences/arrays are a fundamental data structure;
- With no native support in XML, they are mapped using one nested element for each array element;
- Some arbitrary element name has to be used;
- Corresponding opening and closing tags per array element results in much wasted space and parsing overhead;

```
<welcome-file-list>
    <welcome-file>index.html</welcome-file>
    <welcome-file>index.htm</welcome-file>
    <welcome-file>index.jsp</welcome-file>
</welcome-file-list>
```

Problems with XML for data serialization: maps

- Maps/dictionaries/hashes are becoming important data structures, namely in dynamic languages.
- Maps are sets of key-value pairs;
- Problems in XML:
    - attributes are very limited, as they cannot hold arbitrary values;
    - this leads to using nested elements (but these are ordered);
- Key-value pairs are mapped to pairs of nested elements:

```
<member>
  <key>ai</key>
  <value>application/postscript</value>
</member>
```

YAML and JSON

- XML weakness for data serialization motivated alternative proposals (e.g., ConciseXML);
- Dynamic languages communities (Python, Perl, Ruby) defined YAML, a text based format appropriate for data serialization;
- With JavaScript, JSON became a popular text-based format;
- YAML 1.2 was standardized as a superset of JSON;
- Some YAML design goals were:
    - human friendly;
    - good interaction with dynamic languages;
    - native support for the more common data structures;
    - allow stream based processing;

YAML/JSON vs XML

Goals:

- XML descends from SGML, with emphasis on representing structured documents;
- YAML/JSON were designed from scratch for data serialization;

Information model:

- an XML document is essentially a tree tagged with attributes;
  - each element contains an unordered list of attribute-value pairs;
  - and an ordered list of child elements;
- YAML/JSON uses data structure constructors: arrays, maps and scalars (e.g., numbers, strings, booleans),

YAML/JSON: arrays

Native array support in YAML/JSON:

- YAML:

```
- index.html
- index.htm
- index.jsp
```

- YAML/JSON:

```
["index.html", "index.htm", "index.jsp"]
```

YAML/JSON: maps

Native array support in YAML/JSON:

- YAML:

```
abs: audio/x-mpeg
aif: audio/x-aiff
aifc: audio/x-aiff
aiff: audio/x-aiff
```

- YAML/JSON:

```
{ "abs": "audio/x-mpeg", "aif": "audio/x-aiff",
  "aifc": "audio/x-aiff", "aiff": "audio/x-aiff" }
```

YAML/JSON vs XML: arrays

YAML:

```
welcome-file-list:
  - index.html
  - index.htm
  - index.jsp
```

YAML/JSON:

```
{ "welcome-file-list":
    ["index.html",
     "index.htm",
     "index.jsp"] }
```

XML:

```
<welcome-file-list>
    <file>index.html</file>
    <file>index.htm</file>
    <file>index.jsp</file>
</welcome-file-list>
```

YAML/JSON vs XML: maps

YAML/JSON:

```
{mime-mappings:
  {"abs": "audio/x-mpeg",
   "aif": "audio/x-aiff",
   "aifc": "audio/x-aiff",
   "aiff": "audio/x-aiff"}}
```

XML:

```
<mime-mapping>
    <key>abs</key>
    <value>audio/x-mpeg</value>
</mime-mapping>
<mime-mapping>
    <key>aif</key>
    <value>audio/x-aiff</value>
</mime-mapping>
<mime-mapping>
    <key>aifc</key>
    <value>audio/x-aiff</value>
</mime-mapping>
<mime-mapping>
    <key>aiff</key>
    <value>audio/x-aiff</value>
</mime-mapping>
```

Filters and filter composition

- A filter is an operation to write/read a data type to/from a stream;
- For composite types, invoke filter for each component;
- XDR filters are bidirectional: single function works both ways;

```
class A implements Streamable{
  int i;
  float f;
  void readFrom(ObjInputStream str) {
    this.i = str.readInt();
    this.f = str.readFloat();
  }
  void writeTo(ObjOutputStream str) {
    str.writeInt(this.i);
    str.writeFloat(this.f);
  }
}
```

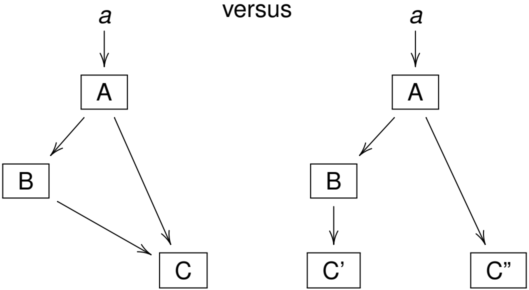
Recursive structures

- To serialize recursive structures (e.g. lists, trees) references must be handled;
- Can be done writing boolean, followed by pointed structure serialization if non-null;
- A generic reference processing operation can be defined;

```
public class ObjOutputStream extends DataOutputStream {
    public void writePtr(Streamable v) throws IOException {
        if (v != null) {
            this.writeBoolean(true);
            v.writeTo(this);
        } else
            this.writeBoolean(false);
    }
public class ObjInputStream extends DataInputStream {
    public Streamable readPtr(Class c) throws IOException {
        if (this.readBoolean())
          return read(c);
        else
          return null;
    }
}
```

Object graphs with sub-structure sharing and cycles

- Previous strategy does not handle sharing and cycles;
- Can be handled encoding index to previously encoded objects;
- Must keep map of encoded/decoded objects;



Polymorfism: encoding type information

- If we know type of objects (fixed class), no need to encode;
- To allow subtype polymorphism, need to encode concrete type (e.g., class name in a string);
- Allows receiver to instantiate object and ask it to read itself;

```
public class ObjInputStream extends DataInputStream {
    // ...
    public Streamable readPoly() throws IOException {
        String s = this.readUTF();
        // ...
        v = (Streamable)Class.forName(s).newInstance();
        // ...
        v.readFrom(this);
        return v;
    }
}
```