An introduction to Apache Avro

Douglas Creager
RedJack, LLC

Mil-OSS WG3
August 30, 2011

REDJACK

What is it?

A framework for:

- *Modeling* your data
- *Serializing* your data
- Making *remote procedure calls*

What is it?

A framework for:

- *Modeling* your data
- *Serializing* your data
- ~~Making *remote procedure calls*~~

Benefits

- Intuitive modeling paradigm
- Efficient binary encoding
- Forwards- and backwards-compatibility via *schema resolution*

Modeling paradigms

Modeling paradigms

Relational model

PEOPLE		
person_id	name	dob
1	Buford	11/19/1953
2	Wilhelmina	04/27/1956
3	Sanjay	06/12/1980
4	Jill	03/08/1982

PARENTS	
parent	child
1	3
2	3
1	4
2	4

Modeling paradigms

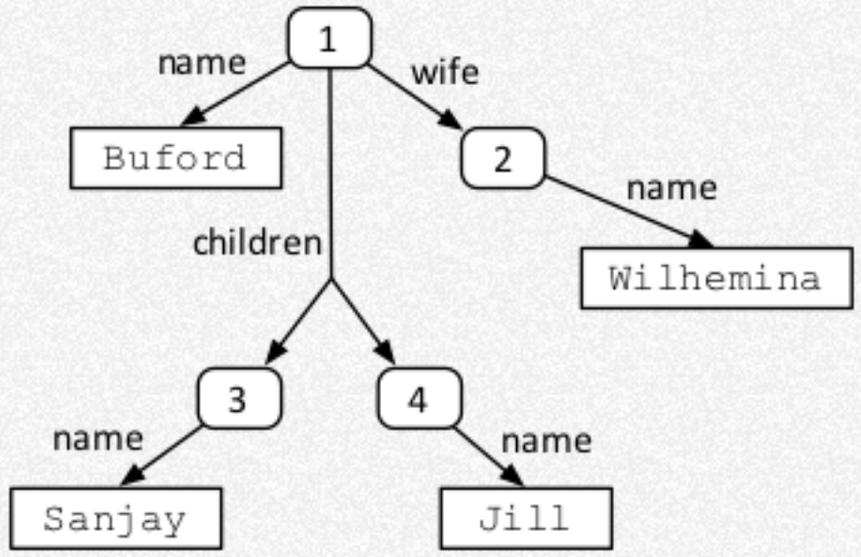
XML – Extensible Markup Language

```
<?xml version="1.0">
<people>
  <person id="1">
    <name>Buford</name>
    <wife>
      <person id="2">
        <name>Wilhelmina</name>
      </person>
    </wife>
    <children>
      <person id="3">
        <name>Sanjay</name>
      </person>
      <person id="3">
        <name>Jill</name>
      </person>
    </children>
  </person>
</people>
```

Modeling paradigms

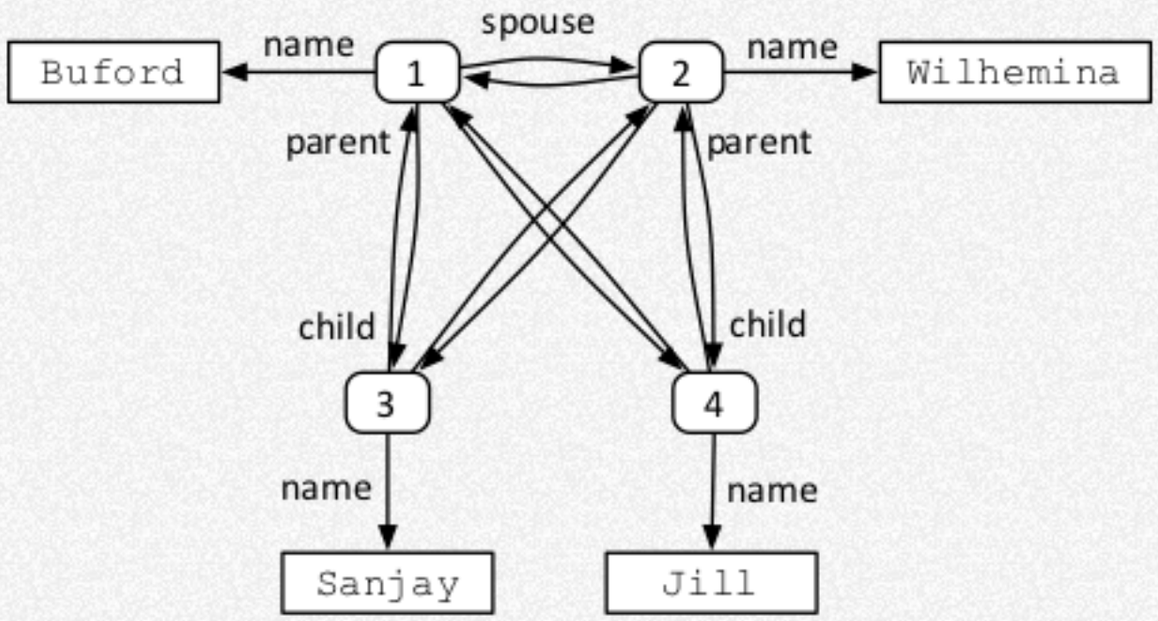
XML – Extensible Markup Language

```
<?xml version="1.0">
<people>
  <person id="1">
    <name>Buford</name>
    <wife>
      <person id="2">
        <name>Wilhelmina</name>
      </person>
    </wife>
    <children>
      <person id="3">
        <name>Sanjay</name>
      </person>
      <person id="3">
        <name>Jill</name>
      </person>
    </children>
  </person>
</people>
```



Modeling paradigms

Semantic Web



Impedence mismatch!

What data structures do we use in our programs?

Impedence mismatch!

What data structures do we use in our programs?

- *scalars*
  - integers (0, 12, -1, 65,536)
  - floats (0.0, 43.12, 1.234e-17)
  - strings (`"Buford"`)
  - booleans (true, false)
  - null

Impedence mismatch!

What data structures do we use in our programs?

- *scalars*
  - integers (0, 12, -1, 65,536)
  - floats (0.0, 43.12, 1.234e-17)
  - strings (`"Buford"`)
  - booleans (true, false)
  - null
- *arrays* (lists, sequences, vectors)

Impedence mismatch!

What data structures do we use in our programs?

- *scalars*
  - integers (0, 12, -1, 65,536)
  - floats (0.0, 43.12, 1.234e-17)
  - strings (`"Buford"`)
  - booleans (true, false)
  - null
- *arrays* (lists, sequences, vectors)
- *maps* (dictionaries, hashes, associative arrays)

Impedence mismatch!

What data structures do we use in our programs?

- *scalars*
  - integers (0, 12, -1, 65,536)
  - floats (0.0, 43.12, 1.234e-17)
  - strings (`"Buford"`)
  - booleans (true, false)
  - null
- *arrays* (lists, sequences, vectors)
- *maps* (dictionaries, hashes, associative arrays)
- *records* (structs, objects, tuples)

Modeling paradigms

JSON – JavaScript Object Notation

```
{
  "id": 1,
  "name": "Buford",
  "wife": {
    "id": 2,
    "name": "Wilhelmina"
  },
  "children": [
    {
      "id": 3,
      "name": "Sanjay"
    },
    {
      "id": 4,
      "name": "Jill"
    }
  ]
}
```

- Uses same data structures as our programs

Serialization and compatibility

Serialization

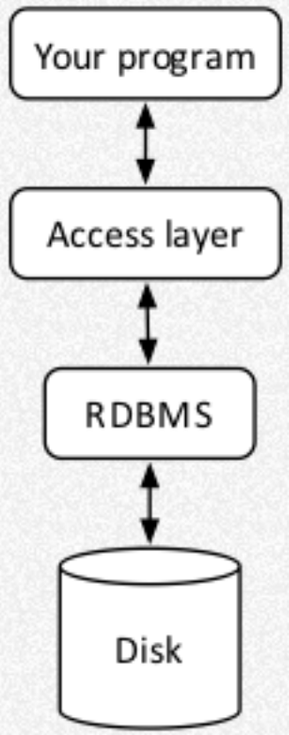
Custom binary formats

- Incredibly efficient
- Even more incredibly difficult to maintain
  - Cumbersome to write
  - Keeping different implementations in sync
  - Forwards and backwards compatibility? Feh.

Serialization

Relational databases

- A great solution:
  You don't care!

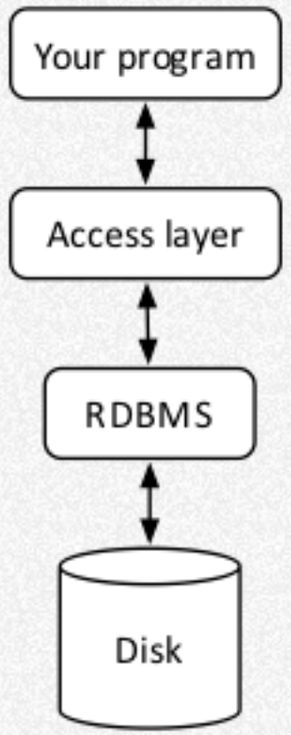


Serialization

Relational databases

- A great solution:
  You don't care!

- But...
  you lose interoperability



Serialization

XML to the rescue!

```
<?xml version="1.0">
<people>
  <person id="1">
    <name>Buford</name>
    <wife>
      <person id="2">
        <name>Wilhelmina</name>
      </person>
    </wife>
    <children>
      <person id="3">
        <name>Sanjay</name>
      </person>
      <person id="3">
        <name>Jill</name>
      </person>
    </children>
  </person>
</people>
```

- Human-readable, standardized syntax
- Let `libxml`/`expat`/whatever handle the parsing

Serialization

XML to the rescue!

```
<?xml version="1.0">
<people>
  <person id="1">
    <name>Buford</name>
    <age>57</age>
    <wife>
      <person id="2">
        <name>Wilhelmina</name>
        <age>55</age>
      </person>
    </wife>
    <children>
      <person id="3">
        <name>Sanjay</name>
        <age>31</age>
      </person>
      <person id="3">
        <name>Jill</name>
        <age>29</age>
      </person>
    </children>
  </person>
</people>
```

- Human-readable, standardized syntax
- Let `libxml`/`expat`/whatever handle the parsing
- Compatibility:
  New tags don't break the parsing

Serialization

“Human-readable”?

```
<?xml version="1.0"?>

<!-- root element wsdl:definitions defines set of related services -->
<wsdl:definitions name="EndorsementSearch"
  targetNamespace="http://namespaces.snowboard-info.com"
  xmlns:es="http://www.snowboard-info.com/EndorsementSearch.wsdl"
  xmlns:esxsd="http://schemas.snowboard-info.com/EndorsementSearch.xsd"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">

  <!-- wsdl:types encapsulates schema definitions of communication types; here
using xsd -->
  <wsdl:types>

    <!-- all type declarations are in a chunk of xsd -->
    <xsd:schema targetNamespace="http://namespaces.snowboard-info.com"
      xmlns:xsd="http://www.w3.org/1999/XMLSchema">

      <!-- xsd definition: GetEndorsingBoarder [manufacturer string, model
string] -->
      <xsd:element name="GetEndorsingBoarder">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="manufacturer" type="string"/>
            <xsd:element name="model" type="string"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
```

From http://www.w3.org/2001/03/14-annotated-WSDL-examples.html

Serialization

“Human-readable”?

```
      <!-- xsd definition: GetEndorsingBoarderResponse [... endorsingBoarder
string ...] -->
      <xsd:element name="GetEndorsingBoarderResponse">
        <xsd:complexType>
          <xsd:all>
            <xsd:element name="endorsingBoarder" type="string"/>
          </xsd:all>
        </xsd:complexType>
      </xsd:element>

      <!-- xsd definition: GetEndorsingBoarderFault [... errorMessage string ...]
-->
      <xsd:element name="GetEndorsingBoarderFault">
        <xsd:complexType>
          <xsd:all>
            <xsd:element name="errorMessage" type="string"/>
          </xsd:all>
        </xsd:complexType>
      </xsd:element>

    </xsd:schema>
  </wsdl:types>

  <!-- wsdl:message elements describe potential transactions -->

  <!-- request GetEndorsingBoarderRequest is of type GetEndorsingBoarder -->
  <wsdl:message name="GetEndorsingBoarderRequest">
    <wsdl:part name="body" element="esxsd:GetEndorsingBoarder"/>
  </wsdl:message>
```

From http://www.w3.org/2001/03/14-annotated-WSDL-examples.html

Serialization

“Human-readable”?

```
  <!-- response GetEndorsingBoarderResponse is of type
GetEndorsingBoarderResponse -->
  <wsdl:message name="GetEndorsingBoarderResponse">
    <wsdl:part name="body" element="esxsd:GetEndorsingBoarderResponse"/>
  </wsdl:message>

  <!-- wsdl:portType describes messages in an operation -->
  <wsdl:portType name="GetEndorsingBoarderPortType">

    <!-- the value of wsdl:operation eludes me -->
    <wsdl:operation name="GetEndorsingBoarder">
      <wsdl:input message="es:GetEndorsingBoarderRequest"/>
      <wsdl:output message="es:GetEndorsingBoarderResponse"/>
      <wsdl:fault message="es:GetEndorsingBoarderFault"/>
    </wsdl:operation>
  </wsdl:portType>

  <!-- wsdl:binding states a serialization protocol for this service -->
  <wsdl:binding name="EndorsementSearchSoapBinding"
                type="es:GetEndorsingBoarderPortType">

    <!-- leverage off soap:binding document style @@@(no wsdl:foo pointing at the
soap binding) -->
    <soap:binding style="document"
                  transport="http://schemas.xmlsoap.org/soap/http"/>

    <!-- semi-opaque container of network transport details classed by
soap:binding above @@@ -->
    <wsdl:operation name="GetEndorsingBoarder">
```

From http://www.w3.org/2001/03/14-annotated-WSDL-examples.html

Serialization

“Human-readable”?

```
      <!-- again bind to SOAP? @@@ -->
      <soap:operation soapAction="http://www.snowboard-
info.com/EndorsementSearch"/>

      <!-- furthur specify that the messages in the wsdl:operation
"GetEndorsingBoarder" use SOAP? @@@ -->
      <wsdl:input>
        <soap:body use="literal"
                   namespace="http://schemas.snowboard-
info.com/EndorsementSearch.xsd"/>
      </wsdl:input>
      <wsdl:output>
        <soap:body use="literal"
                   namespace="http://schemas.snowboard-
info.com/EndorsementSearch.xsd"/>
      </wsdl:output>
      <wsdl:fault>
        <soap:body use="literal"
                   namespace="http://schemas.snowboard-
info.com/EndorsementSearch.xsd"/>
      </wsdl:fault>
    </wsdl:operation>
  </wsdl:binding>

  <!-- wsdl:service names a new service "EndorsementSearchService" -->
  <wsdl:service name="EndorsementSearchService">
    <wsdl:documentation>snowboarding-info.com Endorsement
Service</wsdl:documentation>
```

From http://www.w3.org/2001/03/14-annotated-WSDL-examples.html

Serialization

“Human-readable”?

```
    <!-- connect it to the binding "EndorsementSearchSoapBinding" above -->
    <wsdl:port name="GetEndorsingBoarderPort"
               binding="es:EndorsementSearchSoapBinding">

      <!-- give the binding an network address -->
      <soap:address location="http://www.snowboard-info.com/EndorsementSearch"/>
    </wsdl:port>
  </wsdl:service>

</wsdl:definitions>
```

From http://www.w3.org/2001/03/14-annotated-WSDL-examples.html

Serialization

“Human-readable”?

The equivalent in C:

```
const char *
GetEndorsingBoarder(const char *manufacturer,
                    const char *model);
```

Serialization

"Human-readable"?

The equivalent in C:

```
const char *
GetEndorsingBoarder(const char *manufacturer,
                    const char *model);
```

- Less and less human-readable with complex types
- *So much* overhead...

Apache Avro (finally)

Data modeling

```
{
  "type": "record",
  "name": "person",
  "fields": [
    {"name": "name",
     "type": "string"},
    {"name": "age",
     "type": "int"},
    {"name": "spouse",
     "type": ["null", "person"]},
    {"name": "children",
     "type": {"type": "array",
              "items": "person"}}
  ]
}
```

Data modeling

```
{
  "type": "record",
  "name": "person",
  "fields": [
    {"name": "name",
     "type": "string"},
    {"name": "age",
     "type": "int"},
    {"name": "spouse",
     "type": ["null", "person"]},
    {"name": "children",
     "type": {"type": "array",
              "items": "person"}}
  ]
}
```

- Schemas are encoded in JSON

Data modeling

```
{
  "type": "record",
  "name": "person",
  "fields": [
    {"name": "name",
     "type": "string"},
    {"name": "age",
     "type": "int"},
    {"name": "spouse",
     "type": ["null", "person"]},
    {"name": "children",
     "type": {"type": "array",
              "items": "person"}}
  ]
}
```

- Schemas are encoded in JSON
- Schema language uses standard programming data structures

Data modeling

```
{
  "type": "record",
  "name": "person",
  "fields": [
    {"name": "name",
     "type": "string"},
    {"name": "age",
     "type": "int"},
    {"name": "spouse",
     "type": ["null", "person"]},
    {"name": "children",
     "type": {"type": "array",
              "items": "person"}}
  ]
}
```

- Schemas are encoded in JSON
- Schema language uses standard programming data structures
- Including:
  - scalars:
    - `boolean`, `double`, `float`, `int`, `long`, `null`, `string`
    - enums
    - variable-size byte arrays
    - fixed-size byte arrays
  - arrays and maps
  - records
  - unions

Serialization

Key assumption:

The writer schema will always be available when you read the data.

Serialization

Scalars

- `int` and `long`: "varint" encoding
  0 → `00`
  -1 → `01`
  1 → `02`
  64 → `80 01`

- `double` and `float`: raw 4- or 8-byte IEEE 754 encoding
- `boolean`: `00` or `01`
- `null`: zero bytes
- `bytes` and `string`:
  varint-encoded length prefix, followed by binary data
- `enum`: encoded as an integer, not a string
- `fixed`: no length prefix needed

Serialization

Compound values

Serialization

Compound values

- `array` and `map`
  - element count followed by elements
  - "blocks" allow you to serialize without knowing full array length in advance

Serialization

Compound values

- `array` and `map`
  - element count followed by elements
  - "blocks" allow you to serialize without knowing full array length in advance
- `union`
  - integer discriminant followed by branch value

Serialization

Compound values

- `array` and `map`
  - element count followed by elements
  - "blocks" allow you to serialize without knowing full array length in advance
- `union`
  - integer discriminant followed by branch value
- `record`
  - field values in order
  - *no field names or indices*
  - no overhead for records

Serialization

```
{
  "name": "Buford",
  "age": 57,
  "wife": {
    "person": {
      "name": "Wilhelmina",
      "age": 55
    }
  },
  "children": [
    {
      "name": "Sanjay",
      "age": 31
    },
    {
      "name": "Jill",
      "age": 29
    }
  ]
}
```

There's also a JSON encoding for human-readability

Serialization

```
{
  "name": "Buford",
  "age": 57,
  "wife": {
    "person": {
      "name": "Wilhelmina",
      "age": 55
    }
  },
  "children": [
    {
      "name": "Sanjay",
      "age": 31
    },
    {
      "name": "Jill",
      "age": 29
    }
  ]
}
```

```
0c Buford                          name
72                               age (57)
02                    wife: branch 1
14 Wilhelmina                      name
6e                               age (55)
04             children: element count
0c Sanjay                          name
3e                               age (31)
08 Jill                            name
3a                               age (29)
00              children: end of array
```

```
0c 42 75 66 6f 72 64 72
02 14 57 69 6c 68 65 6c
6d 69 6e 61 6e 04 0c 53
61 6e 6a 61 79 3e 08 4a
69 6c 6c 3a 00
```

Schema resolution

- Since writer schema is available, reader schema doesn't have to match perfectly
- Unused record fields are skipped
- Missing record fields must have default values
- Fields, branches, and enum symbols can be reordered
- Promotion of scalar values (`int` → `long`, etc)

Project information

- Language bindings: Java, Python, Ruby, C, C++, etc.
- Website: http://avro.apache.org/
- Specification
- Mailing lists