4. Encoding and Evolution

- Evolvability
  - Code changes is not instant
    - Servers may perform rolling upgrade (staged rollout)
    - Clients may continue using old code
  - Old and new code need to interact
    - Backward compatibility
    - Forward compatibility
- Formats for Encoding Data
  - Two representations of data
    - In-memory
    - Encoded to pass it somewhere
      - Can't keep pointers
  - Need translations between two representations
    - encoding (serialization or marshalling)
    - decoding (parsing, deserialization, unmarshalling)
  - Language-Specific Formats
    - Benefits
      - Convenient, due to minimal additional code
    - Problems
      - Tied to specific language
      - ⚠️ Security breaches sometimes when decoding instantiates arbitrary classes
      - Versioning is often afterthought
      - Efficiency is often afterthought
    - So usually it is a and idea to use your language's built-in encoding!
  - JSON, XML, and Binary Variants
    - Text and somewhat human-readable
      - XML
        - too verbose and unnecessarily complicated
      - JSON
      - CSV
      - Problems
        - Ambiguity between strings and numbers and different numbers
          - Float vs Int vs Decimal
        - Not support binary data
          - Base64 as workaround
        - Data schemas exist for XML and JSON but is not widely used
        - CSV doesn't have schema
      - Main advantage
        - Easy to agree on their usage to interchange data between organisations
    - Binary encoding
      - for JSON
        - MessagePack, BSON, BJSON, UBJSON, BISON, and Smile
      - for XML
        - WBXML and Fast Infoset
      - Problem
        - As they don't have schema they still need to encode field names
          - Due to this not a big saving
        - But they lose human readability
  - Thrift and Protocol Buffers
    - Thrift
      - Two different formats
        - BinaryProtocol
        - CompactProtocol
    - Protocol Buffers
    - They both use field tags instead of field names
    - Field tags and schema evolution
      - Forward and backward compatibility because of tags
        - Just can't reuse tags
        - And can't add/remove required fields
    - Datatypes and schema evolution
      - Can change data types
        - Int32 -> int64
      - Protocol Buffers
        - Can change optional to repeated
  - Apache Avro
    - nothing to identify fields or their datatypes
      - encoding simply consists of values concatenated together
    - Reader and writers should use exact same schema?
    - The writer's schema and the reader's schema
      - Avro reader knows writer schema and reader schema - and it analyses them to decide how to parse records
      - Can only add/remove fields that have default values
    - But what is the writer's schema?
      - contexts
        - Large file with lots of records
          - Schema is once at the beginning of the file
        - Database with individually written records
          - Each record has version number of schema
        - Sending records over a network connection
          - Processes can negotiate the schema version on connection setup
    - Dynamically generated schemas
      - Advantage
        - Schema doesn't contain field tags
          - As result it is friendlier to dynamically generated schemas (e.g. from database table)
    - Code generation and dynamically typed languages
      - Avro can be used w/o code generation
        - Which is good for dynamically typed languages
  - The Merits of Schemas
    - binary encodings based on schemas are good
      - More compact
      - Schema is a documentation
      - Database of schemas can be tested for backward/forward compatibility before deploy
      - For statically typed languages code generation from schemas enables type checking at compile time
    - 👍 Schema evolution allows flexibility that gives schema-on-read approach but also provides better guarantees and tooling
- Modes of Dataflow
  - Dataflow Through Databases
    - Single process that access db
      - sending a message to your future self
        - Need backward compatibility
    - Several processes access db
      - older and newer code read and write db
        - Forward compatibility is also required
        - ⚠️ During read-update-write important not to lose fields that you don't know yet
    - Different values written at different times
      - ⚠️ data outlives code
      - Rewriting (migrating) data to new schema
        - Possible
        - But expensive on large datasets
      - So - different rows have different versions of schema
    - Archival storage
      - Can use the same latest schema version for all rows, and data becomes immutable
        - Good for formats like Avro
        - Also good for analytical-friendly formats like Parquet
  - Dataflow Through Services: REST and RPC
    - clients and servers
      - API exposed by the server is known as a "service"
    - javascript code inside web browser can become a client
      - Ajax
    - server can be a client to another server
      - Service-Oriented Architecture (SOA) aka "microservices architecture"
        - 👍 Makes the app easier to change and maintain by making services independently deployable and evolvable
        - ⚠️ data encoding used by servers and clients must be compatible across versions of the service API
    - Web services
      - REST
        - Is a design philosophy that builds on principles of HTTP
        - API designed according to the principles of REST is called RESTful
        - OpenAPI aka Swagger used to describe RESTful API
      - SOAP
        - WSDL used to describe SOAP API
          - Can generate code for server/client
    - 👎 The problems with remote procedure calls (RPCs)
      - Different approaches existed
        - EJB and RMI
        - DCOM
        - CORBA
      - Call remote function like you call it local
        - "location transparency"
      - This approach flawed!
        - Network calls are unpredictable
        - Network calls can return timeout
        - Retries can actually cause multiple calls
          - You should implement deduplication (idempotence)
        - Latency can vary
        - Objects should be encoded
          - Can be a problem for large objects
        - Client and server can be written in different languages
          - Data types can be different
      - So - don't hide remote calls!
    - 👍 Current directions for RPC
      - New RPC frameworks
        - Thrift and Avro has their RPC support
        - gRPC for ProtoBuf
        - Finagle uses Thrift
        - Rest.li uses JSON over HTTP
      - New features
        - Futures (promises)
          - to encapsulate asynchronous actions that may fail
        - Streams
          - series of requests and responses
        - Service Discovery
      - Custom RPC protocols with a binary encoding
        - Better performance
      - But RESTful API
        - is good for experimentation and debugging
    - Data encoding and evolution for RPC
      - backward and forward compatibility properties
        - inherited from selected encoding
  - Message-Passing Dataflow
    - asynchronous message-passing systems
      - message
      - message broker
        - also called a "message queue" or "message-oriented middleware"
    - 👍 Advantages over RPC
      - can act as a buffer
        - improve system reliability
      - can redeliver messages to a process that has crashed
        - prevent messages from being lost
      - avoids the sender needing to know the IP address and port number of the recipient
      - allows one message to be sent to several recipients
      - logically decouples the sender from the recipient
    - 👎 Message brokers
      - typically don't enforce any particular data model
        - message is just sequence of bytes with some metadata
      - If consumer re-publishes messages to another topic it should take care of unknown fields!
    - Distributed actor frameworks
      - actor model
        - each actor can receive and send messages
        - It can have state
        - It is single threaded
      - Benefits
        - No need to deal with threads (race conditions, deadlocks)
      - Distributed actors used to scale app to multiple nodes
      - Location transparency works better with actors
        - Because actors already assume that message can be lost
      - Most popular frameworks
        - Akka
          - No support for forward/backward compatibility
          - But can use Protobuf
            - To support rolling updates
        - Orleans
          - No support for rolling update
        - Erlang
          - Rolling updates are hard