Designing Data Intensive Applications

CHAPTER 4

Encoding And evolution

As application grows over time, new changes will be introduced in application.

↳ How will you rollout these changes without breaking the application and continue serving customers?

↳ Rolling upgrade or staged rollout.

↳ don't deploy at once. Deploy on one node then validate, Deploy on other nodes and validate and so on.

① Backward compatible → new code can work with older data.

② Forward compatible → Old code can work with newer data.

How programs work with data?

① In memory → data to be accessed by CPU on machine via objects lists, hashmaps, etc.

② Send data over network → encode it to some different format example JSON.

Translation from ① → ② is called encoding/serialization/marshalling

reverse is called Decoding.

How to encode?

① Language specific support
Java - java.io.Serializable
Python - Pickle
- They should not be avoided for any extensive purposes
  - ↳ Encoded in Java will cause problems to decode in Python.
  - ↳ Lack of forward / backward compatibility.

② JSON, XML and Binary variants
- ↳ support across programming languages.

JSON / XML / CSV
- ↳ Can't distinguish between number and string.
- ↳ No support for binary strings. → workaround using Base64.
- ↳ CSV don't have any schema, application define meaning of each row/column.

Binary Encoding
- ↳ very helpful as data size grows.
- ↳ example for JSON - BSON, BJSON, etc.

Thrift and Protocol Buffers
- ↳ Both require schema for data to be encoded.

example Thrift

```
struct Person {
    1: required string username,
    2: optional i64    favouriteNo
}
```

↳ req/optional is runtime check and dont reflect in encoding.

→ Thrift has two binary encoding formats
- ↳ Binary Protocol
- ↳ Compact Protocol. → takes less space.

Schema Evolution
- ↳ schema evolves over time. How will you ensure encoding / decoding is backward / forward compatible.

↳ Every field in encoded data is identified by tag numbers.
  ↳ If field is not set, it is skipped from record.
↳ Field names can be changed, but field tags should not be changed as they are used for encoding data.
↳ Each new field added should be given a new tag number.

Backward compatible

↳ ignore the field (datatype helps in skipping byte offset) which are newly added on read.

Forward Compatible

↳ As we ensure unique tag number for each field, it will not cause any issues.
↳ It should not be made required to avoid runtime errors.

Data Type evolution in schema

↳ value can loose precision or get truncated.

Avro → binary encoding format
↳ use schema to specify structure for data to be encoded.
↳ no tag numbers.

```
record Person {
    string username;
    array<string> interests;
}
```

↳ most compact as compared to others.
↳ Avro defines schema in terms of writer's schema (who writes) and reader's schema (who reads)
↳ It works fine as far as they are compatible.

↳ to add/remove value, field should have default value.

↳ maintain version number as schema evolves.

Dynamically generated schema in Avro

↳ You don't have to worry about tag numbers unlike Thrift.

↳ the schema can be generated from your object schema.

Dataflow

① Dataflow through Databases

↳ database will change over time eg. add new columns to RDBMS

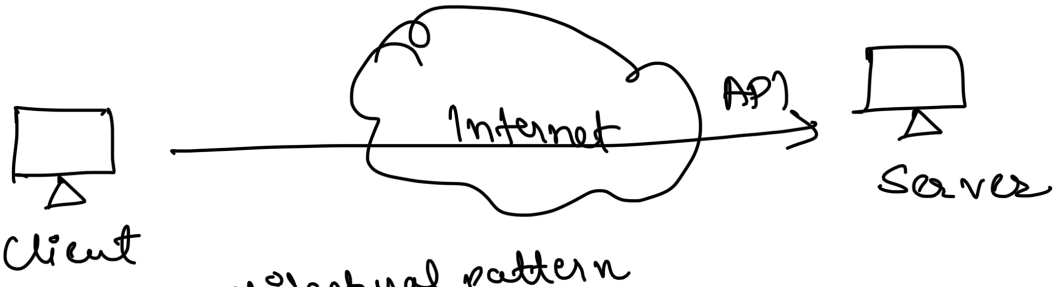
↳ You can specify a default value for newly introduced columns.

↳ Handle specifically in application code as per requirement.

↳ It is good choice as compared to migrating entire db to new schema.

② Dataflow through services – REST & RPC

↳ communication over network.



REST → Design philosophy that builds upon principles of HTTP. (architectural pattern)

SOAP → XML based protocol for network API calls.

eg. GET, PUT

RPC → Remote Procedure call

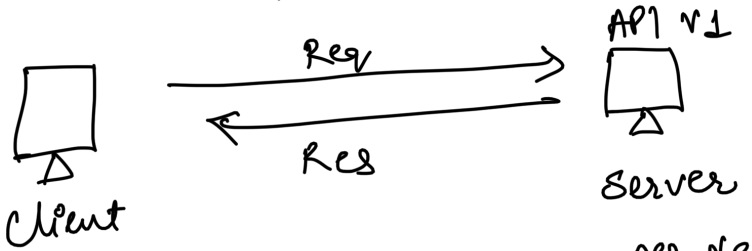
↳ make request to remote network service.

One example- gRPC → support streams, a call can consist of series of request and response over time.

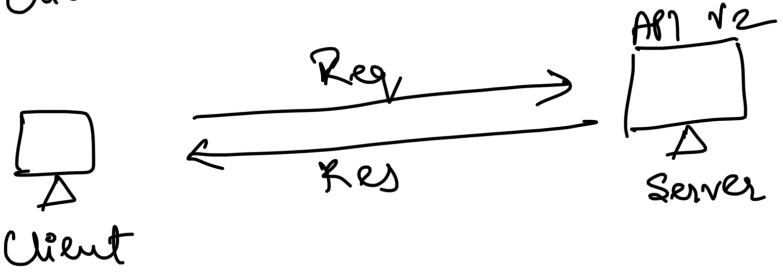
Data evolution → servers will be updated first and then clients

↳ backward compatibity for requests.

↳ forward compatibity for respone.

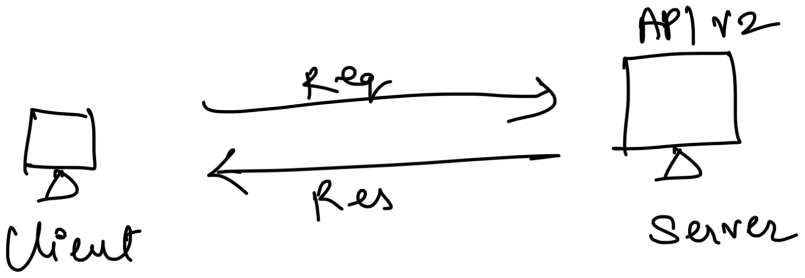


initial system



API updated to v2

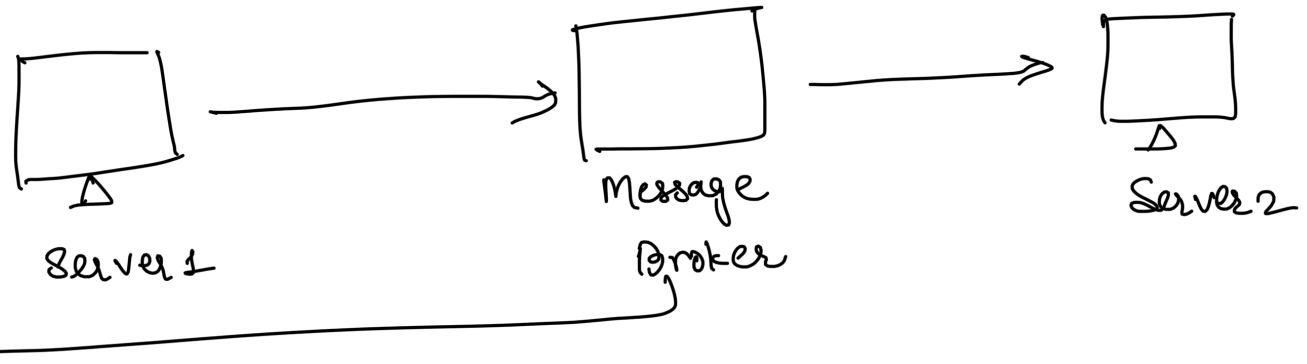
↳ Req is still old, server should handle it.



request updated at client end.

Message Passing Dataflow

↳ asyn processing



↳ temporary storage, can store msg for sometime if receiver is not available → makes system reliable

↳ Receiver Server can consume msgs on its rate.

↳ One msg can be sent to multiple receivers.
↳ Sender doesn't wait for msg to be delivered to receivers.

Example Kafka, RabbitMQ, Amazon SNS-SQS

Actor framework

↳ programming model for concurrency in single process.
↳ Each actor represents one client.
↳ Actors communicate with each other by sending msgs async.

Example
Akka, Erlang OTP

Subscribe for more such content.

YT channel- MsDeep Singh

Happy Learning ☺