Serialization

Technion – Institute of Technology
236700
Software Architecture

• A big software might have a lot of different components which are structured and connected in some ways.
  • E.g. GUI, Server Communication etc.
• Components have strict separation between them.
  • All communication is done via an API.
  • This is either by design
    • Your GUI classes can make SQL queries, but they shouldn’t
  • Or by necessity
    • A remote client cannot access a server’s memory or hardware.

So – how do these components communicate with one another?
API Communication

• API for communication between components is different from “regular” APIs in several key ways:
  • Language heterogeneity
    • It is common for different components to use different languages.
    • E.g. The GUI will be in Java, but the back-end will be C++.
  • Network Communication
    • Different components might even run on different machines, and the communication will be over the web.
  • Team Locality
    • Changes get costlier when communication is harder.
    • “organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations.” –M. Conway
Serialization

• Since components communicate via API and can’t share memory, data has to be serialized when used as parameters or return values.

• There are different kinds of serialization:
  • Binary Serialization
  • Custom Format Serialization
  • Known Format Serialization
  • Automatically Serialized Data
Binary Serialization

- We can use the host language’s serialization ability to convert an object to and from binary data.
  - No need to write any extra code
    - Just make sure everything implements `Serializable`
  - But serialization by reflection can be several orders of magnitude times slower
- No **backward compatibility**
  - If we add or change any data to our classes, we won’t be able to save or reload data.
  - Strict versioning has to be kept in all components.
  - Even different Java versions can trigger failures!
  - Can use `serialVersionUID` to verify the data is taken from the same version as the class.
- Although tempting (easy), almost never the right choice
public class Person implements Serializable {
    public String name;
    public String address;
    public int age;
    public List<Person> children;
}

Person p = new Person();
p.name = "Dor Granat"; p.address = "Technion"; p.age = 21;
p.children = Collections.emptyList();
try {
    FileOutputStream fileOut = new FileOutputStream("/tmp/person");
    ObjectOutputStream out = new ObjectOutputStream(fileOut);
    out.writeObject(p); out.close(); fileOut.close();
} catch (IOException e) { e.printStackTrace(); }
Person p = null;
try {
    FileInputStream fileIn = new FileInputStream("/tmp/person");
    ObjectInputStream in = new ObjectInputStream(fileIn);
    p = (Person) in.readObject();
in.close();
    fileIn.close();
} catch (IOException e) {
    e.printStackTrace();
    return;
}
Custom Format Serialization

- Using your own serialization format to communicate data.
- While the same arguments for custom DSL also hold for custom serialization, custom serialization is usually much less desirable for several reasons:
  - **Data**, unlike **configuration**, doesn’t gain much from controlling the format.
    - We usually just represent a List or Map, so gaining advantage over known formats is rare.
  - If you are after space or speed improvements, it’s very unlikely you’ll beat off-the-shelf serializers.
  - You still have to write your own parser.
Known Format Serialization

- Pretty much any non-esoteric language has a CSV/XML/JSON/YAML parser.
- Which means clients will have no problem consuming your data, if serialized with a known format.
- If your data is flat (e.g. classes with only primitive fields) you can use CSV/TSV.
- But once you start nesting it is time to pull out the “real” formats.
Known Format Serialization

• You still have to decide between writing your own formatter or using reflection.

• Reflection
  • Much slower, especially for nested data (but speed might not be that big of an issue...)
  • Can expose data we don’t care about or want to hide.
  • Most automatic formatters require your data to be mutable (and we don’t like mutations).
  • Usually has an ability to annotate which fields to serialize.

• Manual
  • Developer time consuming
  • Error prone
Serialization Usage

- Regardless of which solution you choose, you should limit serialization and deserialization to the outskirts of your program.
  - Safety
    - A JObject is about as safe as an Object, and more annoying to use.
    - Avoid revalidating data.
  - Format Decoupling
    - Can easily replace format if de/serialization is done in one location only.
Automatically Serialized Data

• We want solutions for all the problems that have arisen from the different methods of de/serialization.
• Binary serialization is not backwards compatible.
• Using reflection is just too slow.
• We don’t want to have multiple implementations of the same data structure
• Solution: define the data structure only once, and use code generation to define implementation in various languages.
Protobufts

• Open source Google library.
• A combination of external (for definition) and internal (for implementation) DSLs.
  • The external DSL lets us define the data structure just once.
    • (actually an SDL - schema definition language)
  • The internal DSL lets us have a quick implementation, not using serialization.
• Extremely efficient binary format.
• Easy backward compatibility.
We define messages. A message is an aggregate containing a set of typed fields.

The “= 1”, “= 2” markers on each element identify the unique “tag” that field uses in the binary encoding.

Each field is annotated with one of:

- **required**: a value for the field must be provided, otherwise the message will be considered "uninitialized".
- **optional**: the field may or may not be set. If an optional field value isn't set, a default value is used.
- **repeated**: the field may be repeated any number of times.
Protobufs: more External DSL

```protobuf
message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;
  enum PhoneType {
    MOBILE = 0;
    HOME = 1;
    WORK = 2;
  }
}
message PhoneNumber {
  required string number = 1;
  optional PhoneType type = 2 [default = HOME];
}
repeated PhoneNumber phones = 4;
}
message AddressBook {
  repeated Person people = 1;
}
```
Protobufts: External DSLs

• As we can see – we also have enums, nested data, etc.

• **required** is forever. If at some point you wish to stop writing or sending a required field, it will be problematic to change to an optional field.
  • Messages without the *required* fields will be rejected or dropped.
  • To ensure flexibility – you can just use only **optional** and **repeated**.

• Tags help us with backward compatibility, because new fields will get new tags. This is transparent for the user, and we can change the order between fields as long as numbers don’t change.
Protobufs: Internal DSLs

- Protobufs exist for most major languages exposing a rich internal DSL.

  // Code generated from the above definition
  // Immutable by default
  Person john = Person.newBuilder()
      .setId(1234)
      .setName("John Doe")
      .setEmail("jdoe@example.com")
      .build();

- Every message type has a class and a builder.
- Every field has getters.
- Repeated fields also have some extra methods.
- Nested enums and messages are generated as nested classes.
External API

As mentioned, components communicate via APIs, but component APIs are:

- Usually very thin, as stability and maintainability are more important than convenience.
- API calls are usually done in a very centralized location in the code, and not all over the place.
- API generally only allows querying, reading, and updating data.
External API: Alternatives

• So, how do we define and access the API?
• One way might be like HTTP parameters:
  https://remoteserver.com/api?user=user-id&media=recent
  • But it can get really out of hand.
• Another common way is REST APIs, which offer alternative to params by encoding them as part of the path:
  https://api.remoteserver.com/users/user-id/media/recent
  • But these are very **stringly typed**.
• There is no API discoverability.
• Honestly – just not very OOP-ish.
RPCs

- Use **proxy objects** (not the proxy design pattern!)
  - Wrap remote external calls with a nice OOP-like API.
  - Remote calls are implemented as methods
    - And call parameters are method parameters
    - Serialization of parameters and deserialization is done for us automatically.
  - This is aptly names **Remote Procedure Call**.
- We can define our services using an external DSL
  - Code generation will create our clients in whatever language we want.
    - Also supporting async or blocking.
  - Ensure correctness of servers by implementing an interface.
We’ll look at gRPC, which conveniently uses ProtobuFs.

- Example taken from the gRPC Java tutorial.

We define services, which contain remote procedure calls.

```java
// The greeting service definition
service Greeter {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloReply) {}  
}

// The request message containing the user’s name.
message HelloRequest {
  string name = 1;
}

// The response message containing the greetings.
message HelloReply {
  string message = 1;
}
```
gRPC – Code Generation

• This external definition will automatically generate gRPC client and server classes.
• However, we still need to implement and call the new methods.
• Let’s have a look at the server implementation:

```java
private class GreeterImpl extends GreeterGrpc.GreeterImplBase {
    @Override
    public void sayHello(HelloRequest req,
                          StreamObserver<HelloReply> responseObserver) {
        HelloReply reply = HelloReply.newBuilder()
                                   .setMessage("Hello " + req.getName())
                                   .build();
        responseObserver.onNext(reply);
        responseObserver.onCompleted();
    }
}
```
This external definition will automatically generate gRPC client and server classes.

However, we still need to implement and call the new methods.

Let’s have a look at the client implementation (a blocking call):

```java
public void greet(String name) {
    HelloRequest request = HelloRequest.newBuilder()
        .setName(name)
        .build();
    HelloReply response;
    try {
        response = blockingStub.sayHello(request);
    } catch (StatusRuntimeException e) {
        return;
    }
    logger.info("Greeting: " + response.getMessage());
}
```