gRPC Basics

Current ONOS: on-platform apps

- Pros:
  - Extremely fast interaction with the ONOS core
  - No possibility of network issues between apps and core
- Cons
  - Lack of isolation for the application
  - Application logic draws from the same resource pool as the ONOS core

Current ONOS: REST apps

- Pros:
  - Enables remote applications to interact with the ONOS core
  - Simple and easy to use
  - Self-descriptive messages
  - Easily human readable
- Cons:
  - Large message sizes due to JSON representation
  - Much slower interaction due to blocking messaging

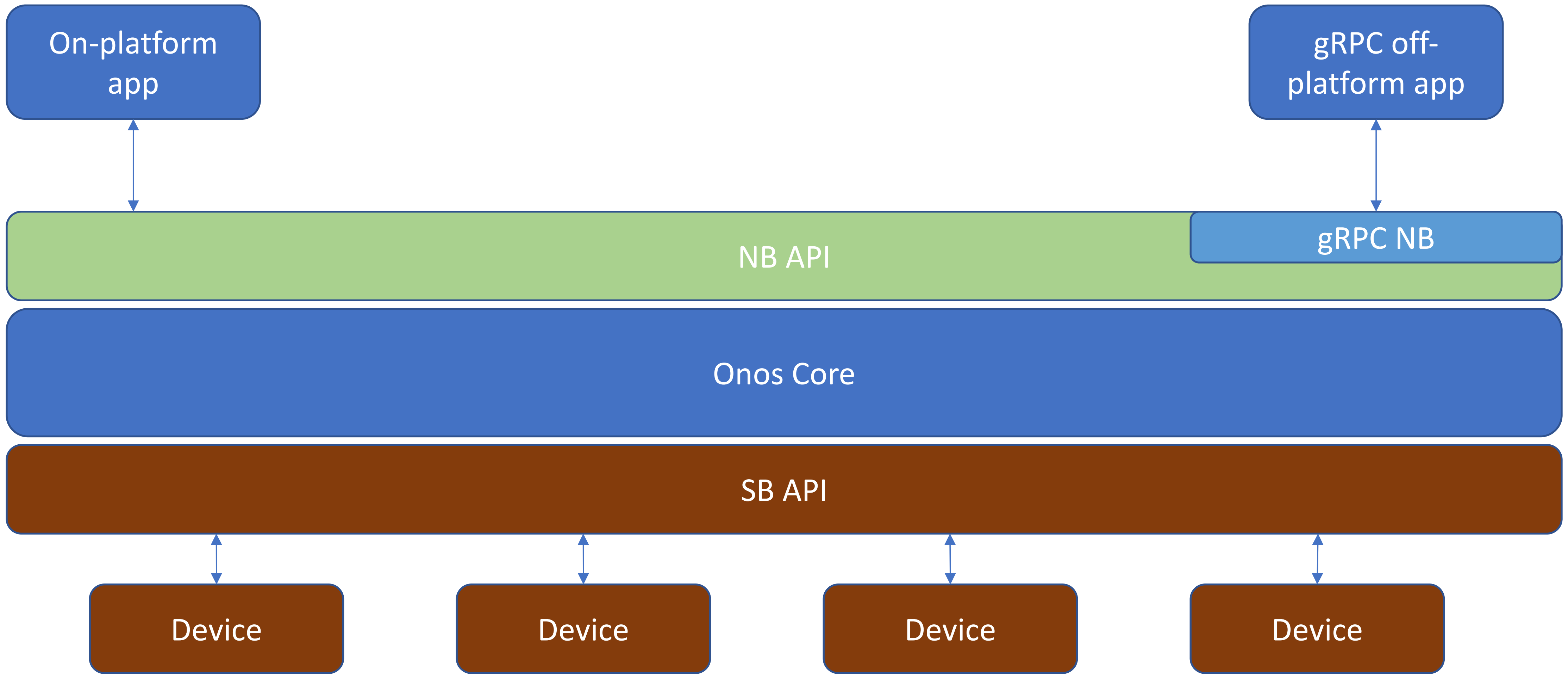
Future ONOS: gRPC

- Pros:
  - Fast non-blocking communication with the ONOS core maintains isolation with less performance penalty
  - Easily human readable messages (can be exported to JSON format)
  - Small message sizes with fast serialization
- Cons:
  - All interacting parties must have message descriptions beforehand
  - Still slower than on-platform

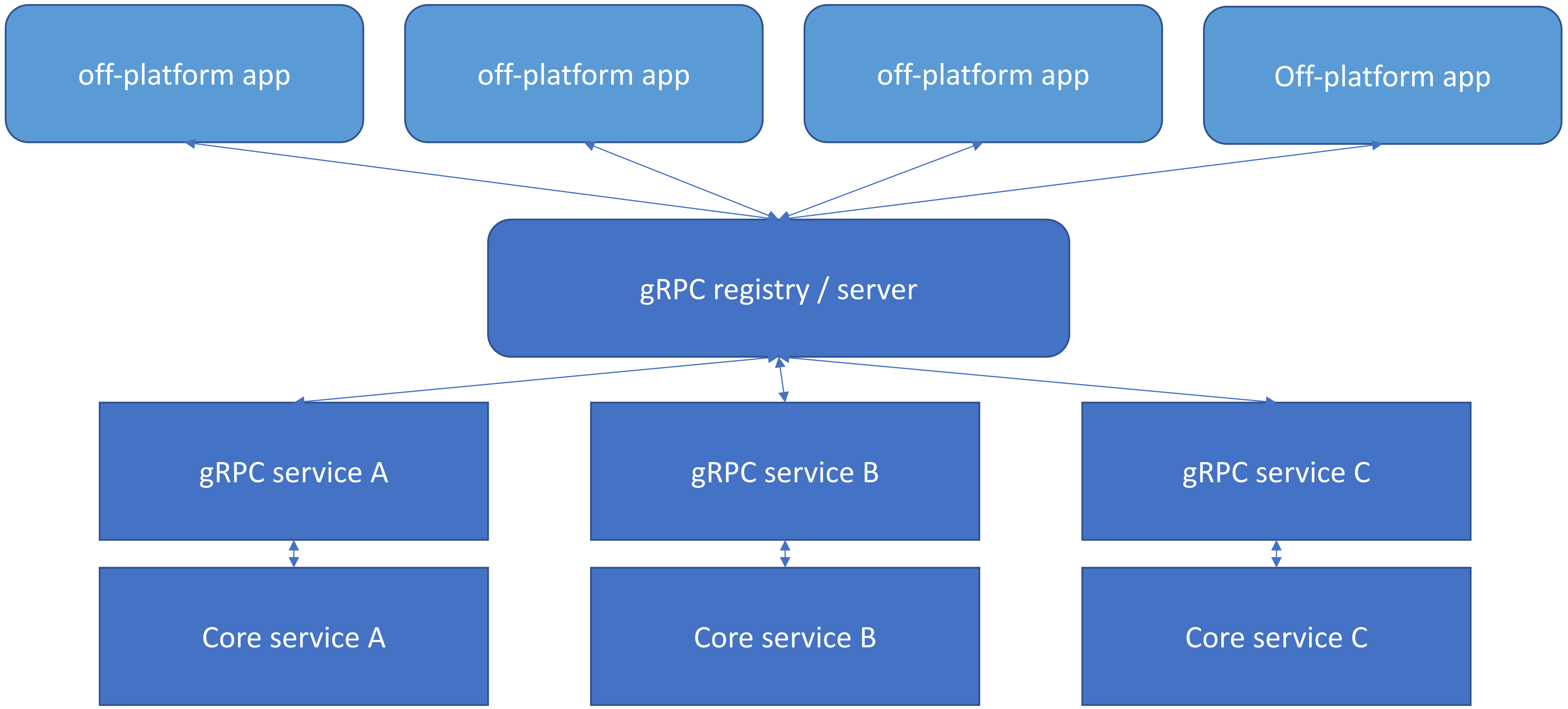
gRPC over http2

- Connections multiplex a single TCP connection avoiding the protocol overhead of multiple handshakes
- Support for bidirectional streaming reduces latency by eliminating blocking calls

How does the gRPC NB fit into ONOS?



How the gRPC subsystem will function



Simple ONOS service in gRPC

- Core API:

```
public class FooService {
        public int foo() {
                …
                return someInt;
        }
        …
}
```

- gRPC analogue:

```
message FooRequest {
}
message FooReply {
        uint32 foo = 1;
}
Service FooServiceGrpc {
        rpc Foo(FooRequest)
                returns (FooReply);
}
```

More complex services

- Core API:

```
public class FooService {
	public Set<foo> multiFoo() {
		…
		return fooSet;
	}
	…
}
```

- gRPC analogue:

```
message MultiFooRequest {
	uint32 num_foos_requested = 1;
}
message MultiFooReply {
	repeated uint32 foo = 1;
}
Service FooServiceGrpc {
	rpc Foo(stream MultiFooRequest)
	returns (stream
		MultiFooReply);
}
```

Using gRPC streams in ONOS

- Advantages:
  - gRPC streaming enables us to transmit objects off-platform that were previously too large
  - May reduce total data transmitted as often only a subset of a large structure is needed
- Challenges:
  - Requires maintaining a snapshot for each pending stream
  - Rapidly streaming a large collection provides little advantage over transmitting all at once

How does gRPC achieve efficiency?

```
message FooReply{
        uint32 foo = 1;
}
```

More specific primitive specification for more efficient encoding

Numerical tags used to identify message fields instead of string parsing

Default values are omitted from the wire format and inferred by the receiver

gRPC Server implementations

- Server implementations should overwrite the methods in the abstract class created by the compilation process
- Each server method will receive a StreamObserver<ResponseType> and a RequestMessage, the StreamObserver can be thought of as belonging to the client thus providing a simple way to pass information to the client
- Stream observers are used to: communicate response messages, notify a client of failure or to terminate a connection
- Server implementations need to create a listening server with a complete listof gRPC service available from that server, additional services cannot be bound once the server has started

gRPC Client Implementations

- Clients create channels to specified servers
- Clients, in certain cases, receive a StreamObserver(although this one observes requests and can be thought of as belonging to the server)
- Clients making non-streaming calls do not require a StreamObserver to call the server and can specify their argument once in the initial method call

gRPC Registry implementation

- It is inefficient to have each gRPC service hosted on its own server so to resolve this problem we will provide a centralized registry which will host all services present in ONOS
- Each service will have an OSGi dependency on the registry guaranteeing that the server begins hosting once the first service is brought online
- Because gRPC servers do not allow services to be added once a server is started initially all services will be brought up at once, in future iterations a modification to gRPC could allow us to dynamically load new services

How gRPC provides backward compatibility?

- Uses the same default value mechanism used to reduce message size
- Older parsers reading new messages ignore new and unrecognized fields although they can be retrieved by slightly more complex means
- Newer parsers infer default values for omitted fields now present in older implementations

Potential gRPC use: ISSU

- ISSU enables a cluster to be updated without losing availability
- Backward compatibility means that updated nodes and old nodes can continue to function together and remain available to commit changes and service queries
- Small message size and fast serialization are beneficial in the high frequency communication among cluster members

gRPC and other languages

- gRPC is language agnostic meaning there is no difficulty communicating between the ONOS core and an off-platform application written in another gRPC supported language (e.g. Python)
- Simple object transmission will be usable without any additional work
- Complex or large object transmission will require some additional support to provide abstractions when streaming data

gRPC language packs

- For other languages there will be language packs which will interact with ONOS gRPC directly and provide reasonable abstractions
- Example ONOS topology:
  - Too large to be transmitted as a single object
  - Can be streamed by providing methods to access nodes by name and by allowing a client to request nodes adjacent to the specified node
  - In order to be effective the client must be able to accept Sets of links and edges and construct a reasonable abstraction of a graph for the remote app to interact with

gRPC automated generation

- We have been exploring the automatic generation of protobuf models and gRPC services from the current ONOS object models
- Automated generation would enable all models to stay in sync and eliminate much of the required maintenance
- The primary issue with automatic generation is tag collisions and custom field reservations

Challenges for gRPC automated generation

- The primary challenge of automated generation is the dependence between the previous version of a file and the current version
  - Without awareness of the previous version or manual management of tag assignment any generation tool would not be able to determine if it was reusing a previously used tag number, such an error would violate backward compatibility
- Secondary challenges include allocation of more desirable tags for user extensions

Demo time!