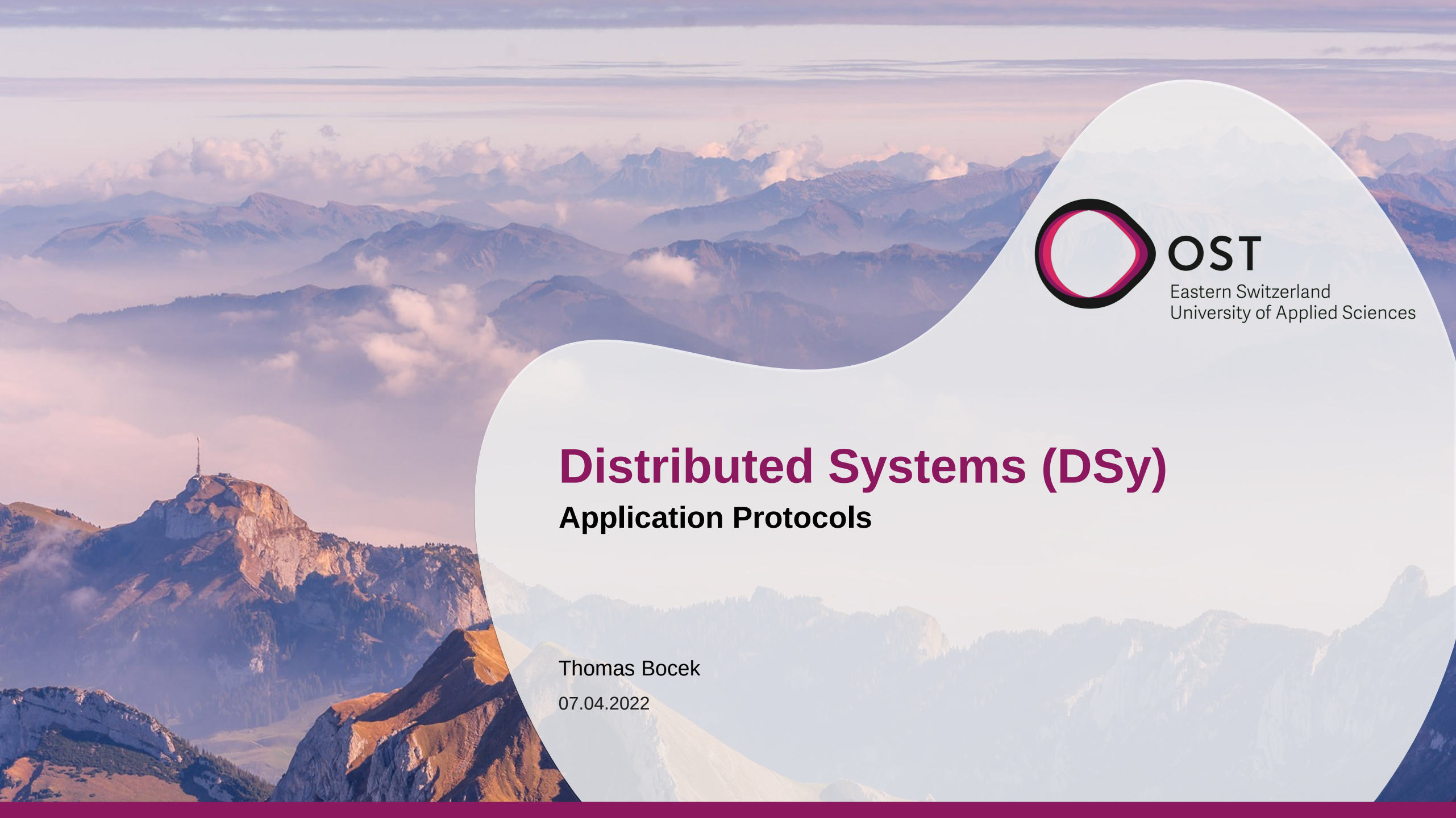


OST
Eastern Switzerland
University of Applied Sciences

Distributed Systems (DSy)

Application Protocols

Thomas Bocek

07.04.2022

Learning Goals

- Lecture 7 (Application Protocols)
  - What protocols besides HTTP exist and why?
  - What are the advantages and how do they work?

Protocols

- Protocols, lecture 3,4: layer 4
  - TCP, UDP, (QUIC/HTTP/3)
- Designing custom protocols (e.g. Kafka)

- Needs more time to develop / test

+ Can be more efficient (space/performance)

- Protocol generators (binary): Thrift / Avro / Protocol Buffers / (ASN1)

+ IDL (interface description language) generates code

+ Standard

- Has more overhead

e.g, Avro IDL - higher-level language for authoring Avro schemata → generates Avro schema

```
//Avro IDL
@namespace("ch.hsr.dsl")

protocol MyProtocol{
  record AMessage {
     string request;
     int code;
  }
  record BMessage {
     string reply;
  }

  BMessage GetMessage(AMessage msg);
}
```

```
{"namespace": "ch.hsr.dsl",
 "type": "record",  "name":
"AMessage",
 "fields": [
  {"name": "request", "type":
"string"},
  {"name": "code", "type": "int"}
 ]
}
```

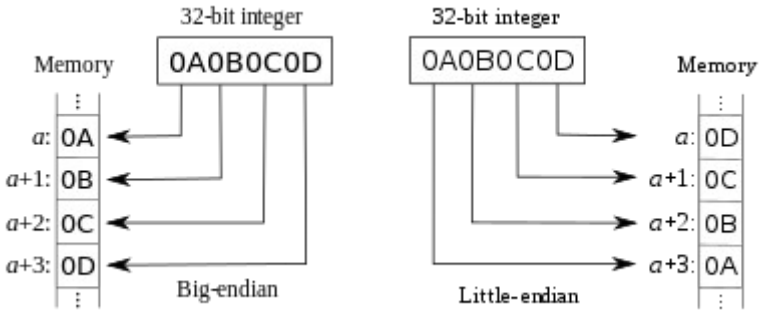

OST

Protocols

- Custom encoding/decoding
  - You control every aspect
  - You send more time on it

```
115     public static boolean decodeHeader(final ByteBuf buffer, final InetSocketAddress recipientSocket,
116             final InetSocketAddress senderSocket, final Message message) {
117         LOG.debug("Decode message. Recipient: {}, Sender:{}.", recipientSocket, senderSocket);
118         final int versionAndType = buffer.readInt();
119         message.version(versionAndType >>> 4);
120         message.type(Type.values()[(versionAndType & Utils.MASK_0F)]);
121         message.protocolType(ProtocolType.values()[versionAndType >>> 30]);
122         message.messageId(buffer.readInt());
123         final int command = buffer.readUnsignedByte();
124         message.command((byte) command);
125         final Number160 recipientID = Number160.decode(buffer);
126
127         //we only get the id for the recipient, the rest we already know
128         final PeerAddress recipient = PeerAddress.builder().peerId(recipientID).build();
129         message.recipient(recipient);
130
131
132         final int contentTypes = buffer.readInt();
133         message.hasContent(contentTypes != 0);
134         message.contentTypes(decodeContentTypes(contentTypes, message));
```

- Little-endian / Big-endian
  - sequential order where bytes are converted into numbers
- Networking, e.g. TCP headers: Big-endian
- Most CPUs e.g., x86: Little-endian, RISC-V: Bi-endianness



[source]

OST

Protocols Examples with Golang

- UDP example in repo DSy
  - Why is it failing?

```
func main() {
   fmt.Println("connecting...")
   conn, _ := net.Dial("udp",
"127.0.0.1:7000")
   defer conn.Close()
   buf := make([]byte, 4)
   binary.LittleEndian.PutUint32(buf, 77)
   conn.Write(buf)
}
```

```
func main() {
   fmt.Println("listening...")
   inet := &net.UDPAddr{net.IPv4zero, 7000, ""}
   udpConn, _ := net.ListenUDP("udp", inet)
   b := make([]byte, 4)
   n, b2, _ := udpConn.ReadFromUDP(b);
   fmt.Printf("connecting... read: %d, addr: %v, data:
%v," +
      " decoded: %v\n", n, b2, b[:n],
binary.BigEndian.Uint32(b))
}
```

- TCP
  - Custom serialization 5,Anybody there?
    - 15 bytes

```
func main() {
   fmt.Println("connecting...")
   conn, _ := net.Dial("tcp", "127.0.0.1:7000")
   defer conn.Close()
   buf := make([]byte, 15)
   buf[0]=5
   copy(buf[1:],[]byte("Anybody there?"))
   _, _ = conn.Write(buf)
}
```

```
func main() {
   fmt.Println("listening...")
   tcpConn, _ := net.Listen("tcp", ":7000")
   conn, _ := tcpConn.Accept() //do this in a go routine
   b := make([]byte, 15)
   n, _ := conn.Read(b)
   fmt.Printf("connecting... read: %d, addr: %v, data: [% x],
decoded: %v\n",
      n, conn.RemoteAddr(), b[:n], string(b[1:]))}
```


OST

Protocols Example ASN1

- ASN1
  - Defined in 1984. Standard interface description language (IDL)
  - Define data structures - can be serialized and deserialized
  - Used e.g., in: X.509 (hsr.ch)
  - Generic binary protocol, Golang package
  - Example: 21 bytes, XML: 48 bytes

```
type TestASN struct {
     Code *big.Int
     Message string
}
…
func main() {
…
   var t TestASN
   _, err = asn1.Unmarshal(b, &t)
}
```

```
30 13 02 01 05 16 0e 41 6e 79 62 6f 64 79 20 74 68 65 72 65 3f
```

```
30 – type tag indicating SEQUENCE
13 – length in octets of value that follows
  02 – type tag indicating INTEGER
  01 – length in octets of value that follows
    05 – value (5)
  16 – type tag indicating IA5String
     (IA5 means the full 7-bit ISO 646 set, including
variants,
      but is generally US-ASCII)
  0e – length in octets of value that follows
    41 6e 79 62 6f 64 79 20 74 68 65 72 65 3f – value
("Anybody there?")
```

```
<code>5</code>
<message>Anybody there?</message>
```


OST

Protocols Example Avro

- Avro: data serialization system
  - Remote procedure call and data serialization framework
  - Use: Hadoop (Big-data framework)
- LinkedIn go-avro library
  - Define a message in JSON (benchmarks) or IDL (slide 12) – no code generation

```
func main() {
  schema, _ := ioutil.ReadFile("schema.avsc")
  codec, _ := goavro.NewCodec(string(schema))
  map1 := map[string]interface{}{
    "request": "Anybody there?",
    "code":  5,
  }
  binary, _ := codec.BinaryFromNative(nil, map1)
  conn, _ := net.Dial("tcp", "127.0.0.1:7000")
  defer conn.Close()
  n, _ := conn.Write(binary)
  fmt.Printf("wrote %d bytes: [% x]\n", n,
binary)
}
```

- Server
  - Example 16 bytes, assuming both have the same IDL

```
{"namespace": "ch.hsr.dsl",
  "type": "record",  "name": "AMessage",
  "fields": [
      {"name": "request", "type": "string"},
      {"name": "code", "type": "int"}
  ]
}
```

```
func main() {
  schema, _ := ioutil.ReadFile("schema.avsc")
  codec, _ := goavro.NewCodec(string(schema))
  tcpConn, _ := net.Listen("tcp", ":7000")
  conn, _ := tcpConn.Accept() //do this in a go routine
  binary := make([]byte, 100)
  n, _ := conn.Read(binary)
  native, _, _ := codec.NativeFromBinary(binary[:n])
  fmt.Printf("read: %v\n", native)
}
```

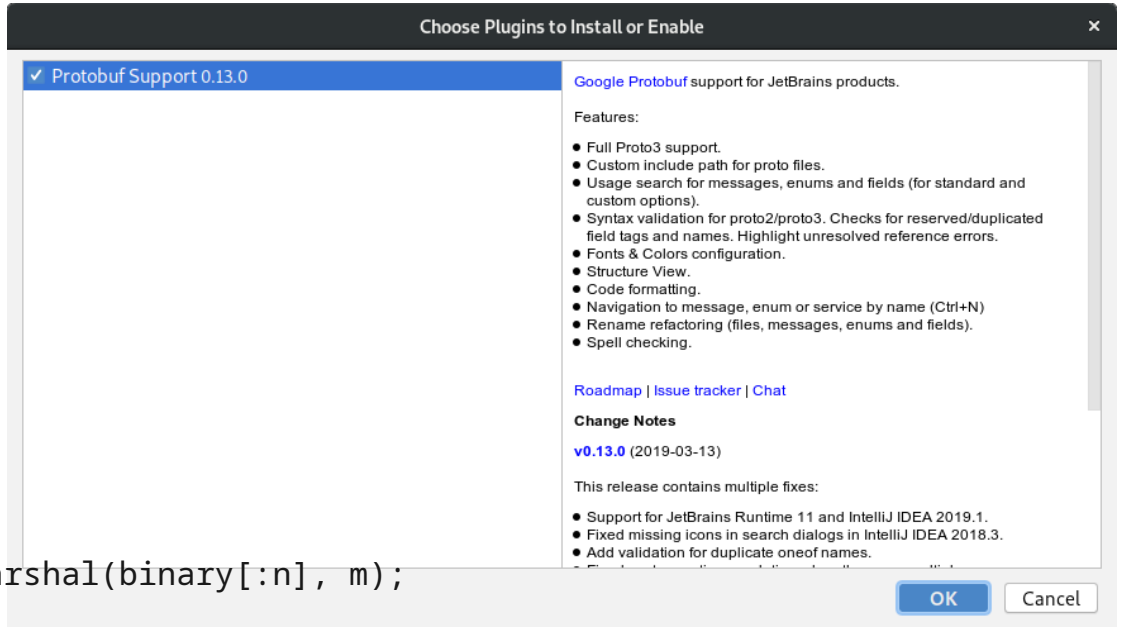

OST

Protocol Buffers Example

- Protobuf: data serialization system from Google
  - Design goals: smaller and faster than XML
  - Use: nearly all inter-machine communication at Google
- Generate 1 go file
  - protoc schema.proto --go_out .
  - integers to identify fields. Protocol buffer contains only numbers, not field names
- 18 bytes

- Not self-describing, but has gzipped description

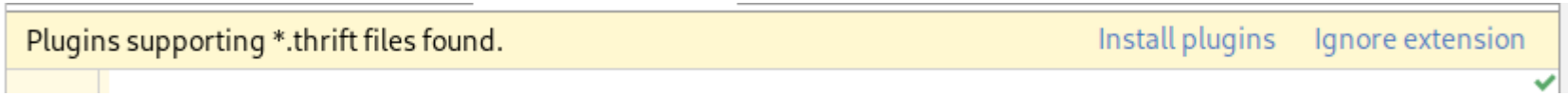
```
syntax = "proto3";
message AMessage {
  int32 code = 1;
  string message = 2;
}
```




```
m := &pb.AMessage{Id: 5, Message: "Anybody
there?"}
out, err := proto.Marshal(m)
```

```
m := &pb.AMessage{}
if err := proto.Unmarshal(binary[:n], m);
err != nil {
  panic(err)
}
```


OST



RPC Example Thrift

```
service TestService {
    void AMessage(1:i32 int, 2:string message)
}
```

- RPC Framework from Facebook
  - IDL and binary protocol
  - For building cross-platform application in ActionScript, C, C++, C#, Cappuccino, Cocoa, Delphi, Erlang, Go, Haskell, Java, Node.js, Objective-C, OCaml, Perl, PHP, Python, Ruby, Rust, Smalltalk, and Swift
- Installation
  - sudo apt install thrift-compiler
  - thrift -r --gen go schema.thrift
  - Creates go files (client)
- Example generic server
  - go run simple-gen-srv.go
  - go run simple-thrift.go -p 7000
  - go run simple-thrift.go -p 7000 AMessage 5 'Anybody there?'
- 49 bytes transferred
- Thrift also encodes which function to call, larger size

OST

RPC Example gRPC

- gRPC
  - Uses HTTP/2 for transport
  - Uses Protocol Buffers
  - Features: authentication, bidirectional streaming and flow control, blocking or nonblocking bindings, and cancellation and timeouts, many languages
  - Installation
    - go get -u google.golang.org/grpc
    - go get -u github.com/golang/protobuf/protoc-gen-go
    - protoc schema-grpc.proto --go_out=plugins=grpc:go-gen3
  - 171 / 124 (wireshark)

```
grpcServer := grpc.NewServer()
s := Server{}
schemagrpc.RegisterMessageServiceServer(grpcSe
rver, &s)
grpcServer.Serve(tcpConn)
```

- Define services and message
  - Generate 1 source file with functions (service)

```
syntax = "proto3";

service MessageService {
  rpc SendMessage (AMessage)
returns(Empty);
}

message AMessage {
  int32 code = 1;
  string message = 2;
}

message Empty {}
```

```
var conn *grpc.ClientConn
conn, _ := grpc.Dial(":7000", grpc.WithInsecure())
if err != nil {
  log.Fatalf("did not connect: %s", err)
}
defer conn.Close()
c := schema_grpc.NewMessageServiceClient(conn)
response, _ := c.SendMessage(context.Background(),
     &schema_grpc.AMessage{Code:5,Message:"Anybody
there?"})
```


OST

JSON example

- JSON + REST/HTTP
  - Human-readable text to transmit data
  - Often used for web apps
- 187 bytes

```
func main() {
  fmt.Println("Connecting...")
  req, _ := http.NewRequest("POST",
"http://localhost:7000",
    strings.NewReader(`{"code": 5,"message": "Anybody
there?"}`))
  req.Header.Set("Content-Type", "application/json")
  client := &http.Client{}
  resp, err := client.Do(req)
  if err != nil {
    panic(err)
  }
  defer resp.Body.Close()
  fmt.Printf("wrote request")
}
```

- Parsing overhead, JSON slower than binary protocol - benchmarks

```
[
    {
        "id": "bitcoin",
        "name": "Bitcoin",
        "symbol": "BTC",
        "rank": "1",
        "price_usd": "9324.08",
        "price_btc": "1.0",
        "24h_volume_usd": "9039300000.0",
        "market_cap_usd": "158560288125",
        "available_supply": "17005462.0",
        "total_supply": "17005462.0",
        "max_supply": "21000000.0",
        "percent_change_1h": "0.46",
        "percent_change_24h": "-0.27",
        "percent_change_7d": "4.5",
        "last_updated": "1525011874“
    }, ...
]
```


OST

Application Protocol: HTTP

- HTTP (HyperText Transfer Protocol): foundation of data communication for www
- Started in 1989 by Tim Berners-Lee
  - HTTP/1.1 published in 1997
  - HTTP/2 published in 2015
    - More efficient, header compression, multiplexing
  - HTTP/3 wip (April 2022: HTTP/3 protocol is an Internet Draft – not yet final)
- Request / response (resource)
- HTTP resources identified by URL
  - https://dsl.hsr.ch/design/hsr_logo.svg

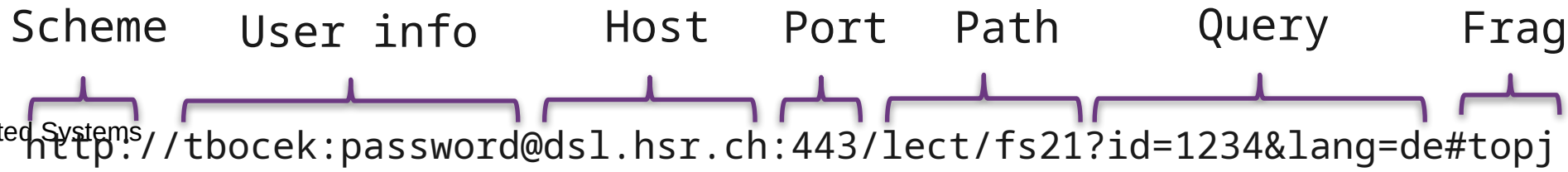
- Text-based protocol

```
openssl s_client -connect dsl.hsr.ch:443
… TLS handshake …
GET /
```

- Browser sends a bit more…

Request Headers (359 B)

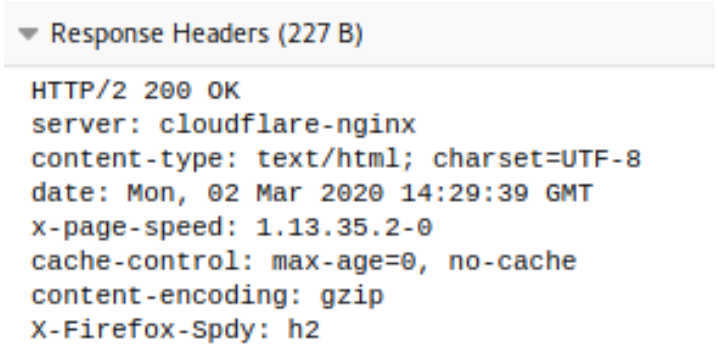
```
Host: dsl.hsr.ch
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:73.0) Gecko/20100101 Firefox/73.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
DNT: 1
Connection: keep-alive
Upgrade-Insecure-Requests: 1
Cache-Control: max-age=0
TE: Trailers
```



OST

Application Protocol: HTTP

- Response
  - Header



  - Status Code: 200
    - List: from 1xx (information response), 2xx (success) – 200 OK, 3xx (redirection), 4xx (client error), 404 Not Found, 403 Forbidden (access slides outside HSR), 5xx (server errors)
  - Content

```
<!DOCTYPE html>
<html>
<head>
    <title>Distributed Systems and Ledgers Lab</title>
    <link rel="stylesheet" type="text/css"
href="design/layout.css"/>
```

- HTTP is a stateless protocol
  - Server maintains no state
- Request methods: GET, HEAD, POST, PUT, DELETE, TRACE, OPTIONS, CONNECT, PATCH
- Web server one-liner - with netcat:
  - while true; do { echo -e "HTTP/1.1 200 OK\n\n<h1> Hallo" } | nc -vl -p 8080 -c; done
- Every Webbrowser has dev tools to show request / responses
  - Firefox, Chrome: ctrl+shift+i / F12
  - Used regularly

OST

Protocols Bencoding and Others

- Benconding
  - Integers: i42e, Byte string: 4:test, list: l4:testi42ee
  - Map/dictionary: d4:test3:hsr3:tomi42ee
- Use: BitTorrent
- UBJSON
- Cap'n Proto , FlatBuffers
  - Do not serialize, just copy, little-endian
- Apache Arrow
  - Do not serialize, copy, and optimally layout for memory access
- … and many others
- Benchmarks, benchmarks, …