OST
Eastern Switzerland
University of Applied Sciences

Distributed Systems & Blockchain (DS1)

Application Protocols

Thomas Bocek

28 March 2021

Protocols

- Protocols, lecture 2: layer 4
  - TCP, UDP, (QUIC)
- 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

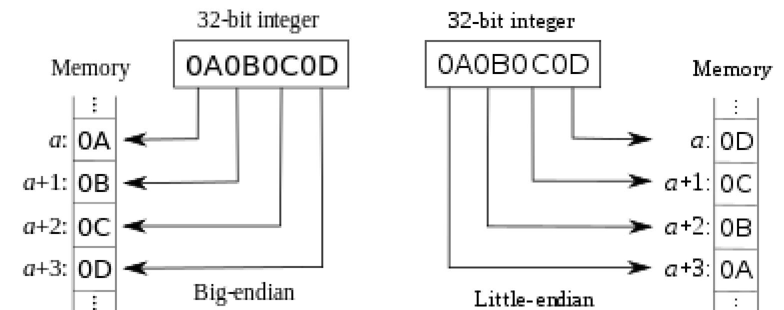
```
//Avro IDL
@namespace("ch.hsr.dsl")

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

```java
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));
```



[source]

Protocols Examples with Golang

- UDP example in repo FS21
  - 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)
}
```

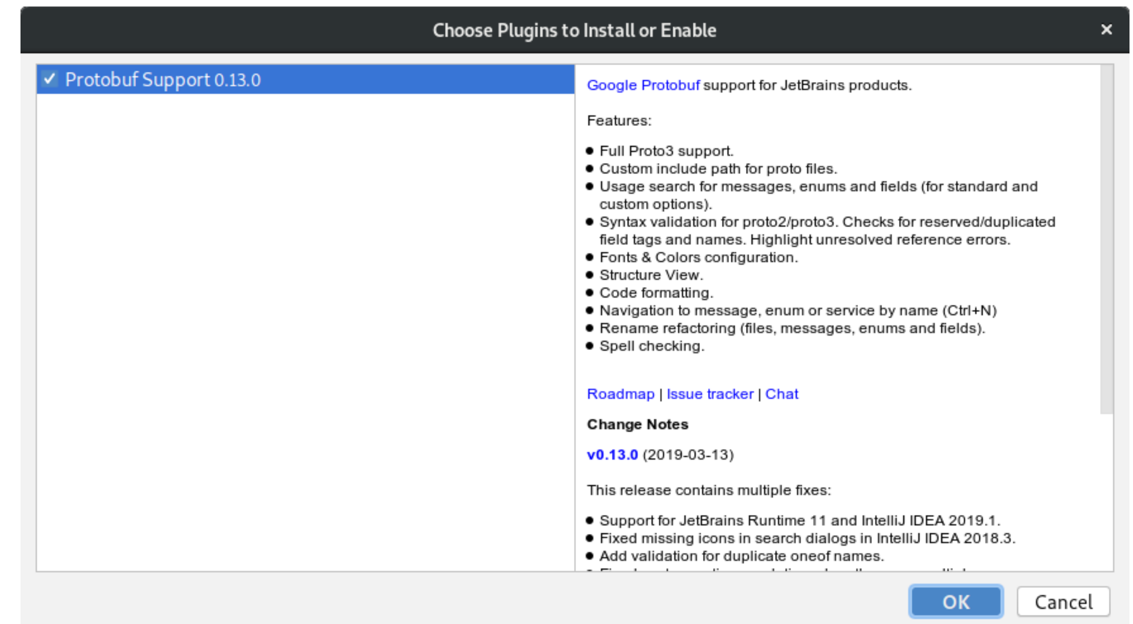
Protocol Buffers Example

- 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

```
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)
}
```

- Not self-describing, but has gzipped description

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



OST

RPC Example Thr

Plugins supporting *.thrift files found. Install plugins Ignore extension

- 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)

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

- 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(grpcServer, &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
  - 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")
}
```

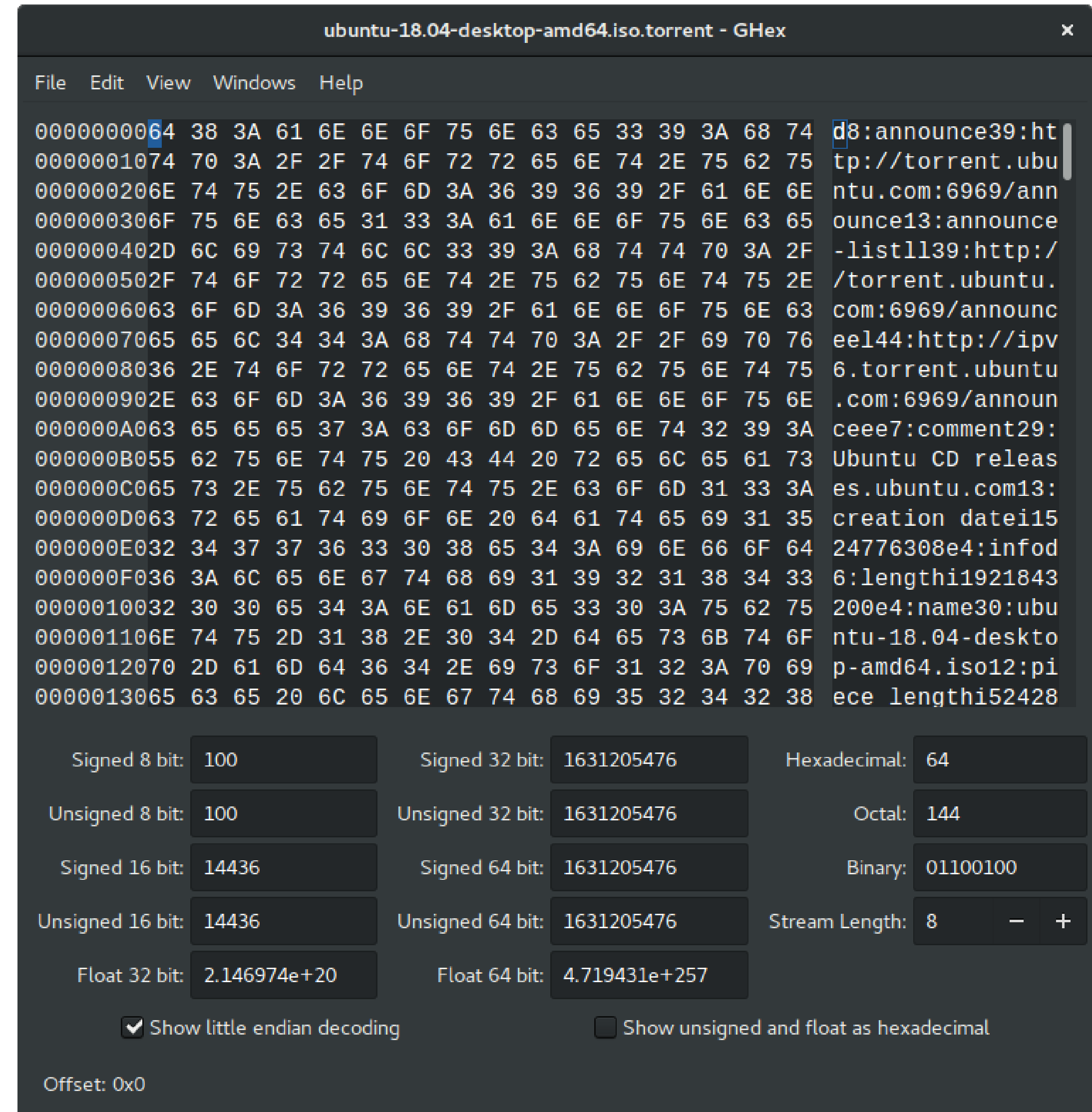
- Parsing overhead, JSON slower than binary protocol - benchmarks


```
[
    {
        "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

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, …



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
- 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 /
```

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
```

Scheme | User info | Host | Port | Path | Query | Fragment

```
http://tbocek:password@dsl.hsr.ch:443/lect/fs21?id=1234&lang=de#topj
```


OST

Application Protocol: HTTP

- Response
  - Header

```
Response Headers (227 B)

HTTP/2 200 OK
server: cloudflare-nginx
content-type: text/html; charset=UTF-8
date: Mon, 02 Mar 2020 14:29:39 GMT
x-page-speed: 1.13.35.2-0
cache-control: max-age=0, no-cache
content-encoding: gzip
X-Firefox-Spdy: h2
```

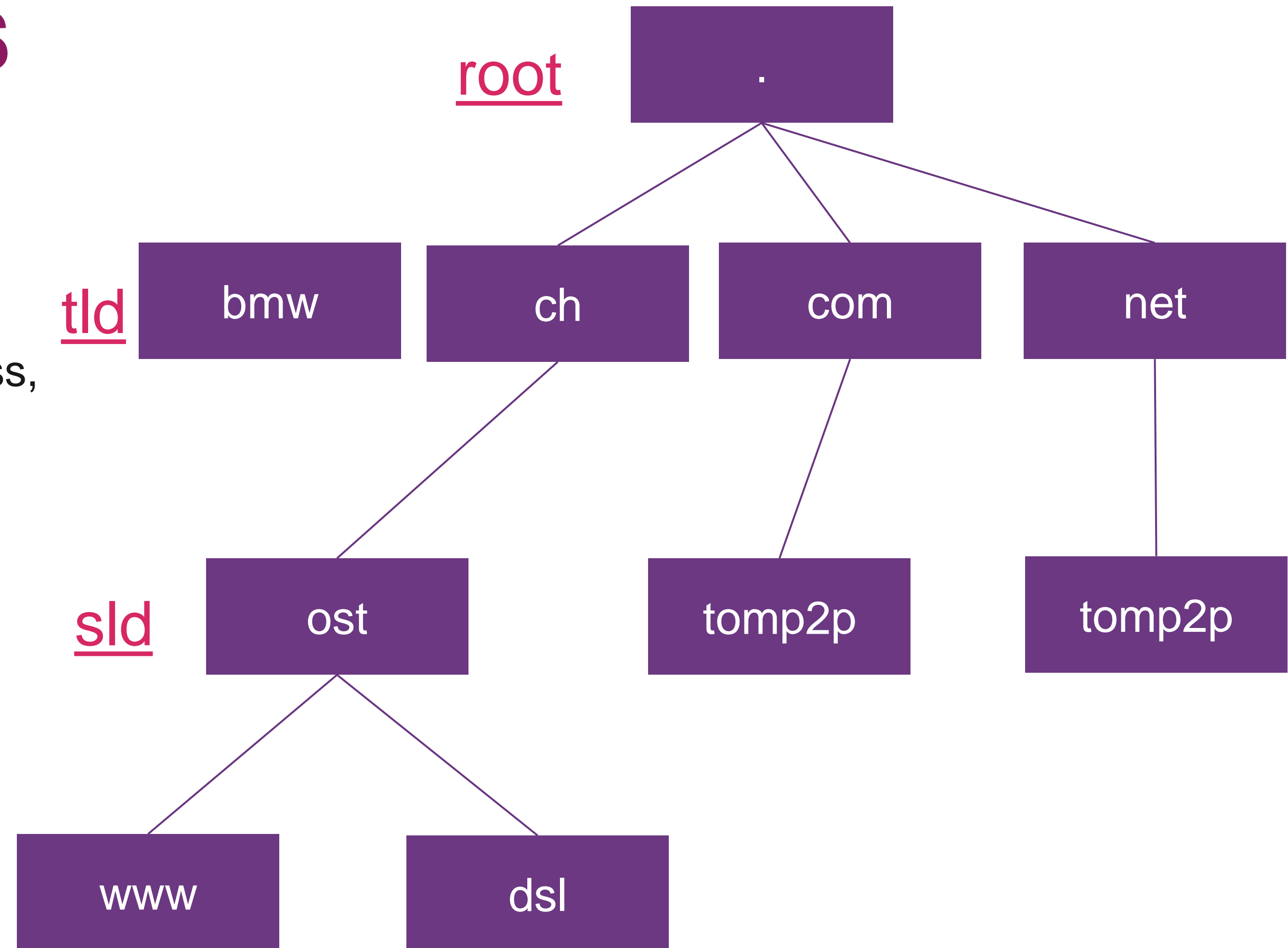
  - Status Code: 200
    - List: from 1xx (information response), 2xx (success) – 200 OK, 3xx (redirection), 4xx (client error), 404 Not Found, 403 Forbidden (access slides outside HSR), 5xx (server errors)
  - Content

```
<!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\r\n\r\n Hallo'; } | nc -l 8080 -q 1; done
- Every Webbrowser has dev tools to show request / responses
  - Firefox, Chrome: ctrl+shift+i / F12
  - Used regularly

Application Protocol: DNS

- Translates human readable domain names to IP addresses “phonebook of the Internet”
  - Delegate authority over sub-domains to other name servers
- Lots of new TLD: .zuerich, .bmw, .americanexpress, .youtube, .გე (application fee 185k USD)
  - No special characters: ASCII (no UTF)
  - Punycode: bücher.tld → xn--bcher-kva.tld
- Hierarchical and decentralized naming system for computers
  - E.g., dsl.hsr.ch
  - Uses UDP, port 53
  - Designed in 1983: unencrypted, unsigned
- Before DNS: exchange of hosts.txt
  - Does not scale



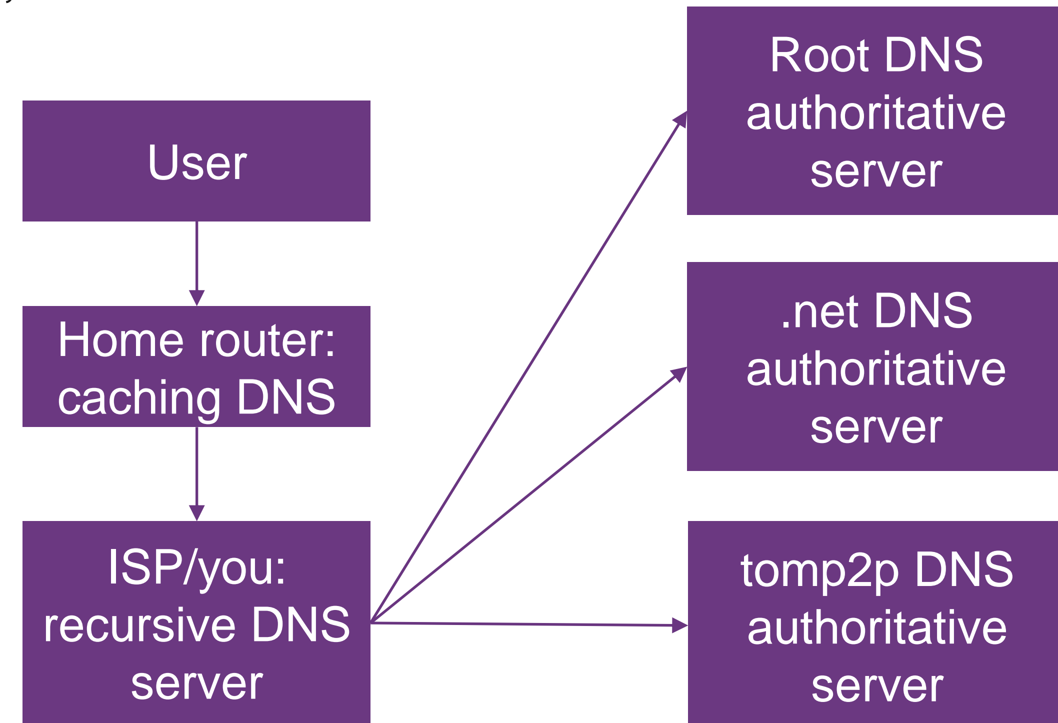
OST

Application Protocol: DNS

- Primary + secondary DNS in case of failure
  - Secondary DNS gets data from primary
- Typical setup
  - User
  - Caching/forwarding DNS (e.g., dnsmasq)
  - Recursive servers: DNS name resolution for applications (e.g, bind/unbound)
  - Authoritative servers: providing a definitive answer of e.g., tomp2p.net (e.g., bind/nsd)
    - Authoritative DNS service allows **others** to find **your** domain; Recursive DNS allows **you** to resolve **other** domains
- Restriction to 13 root servers due to 512 byte packet limit
  - With anycast, ~1000 root servers around the world

E.g. BIND

```
zone tomp2p.net  {
        type master;
        file "zones/tomp2p.net";
        allow-transfer { 192.168.0.3; };
};
```



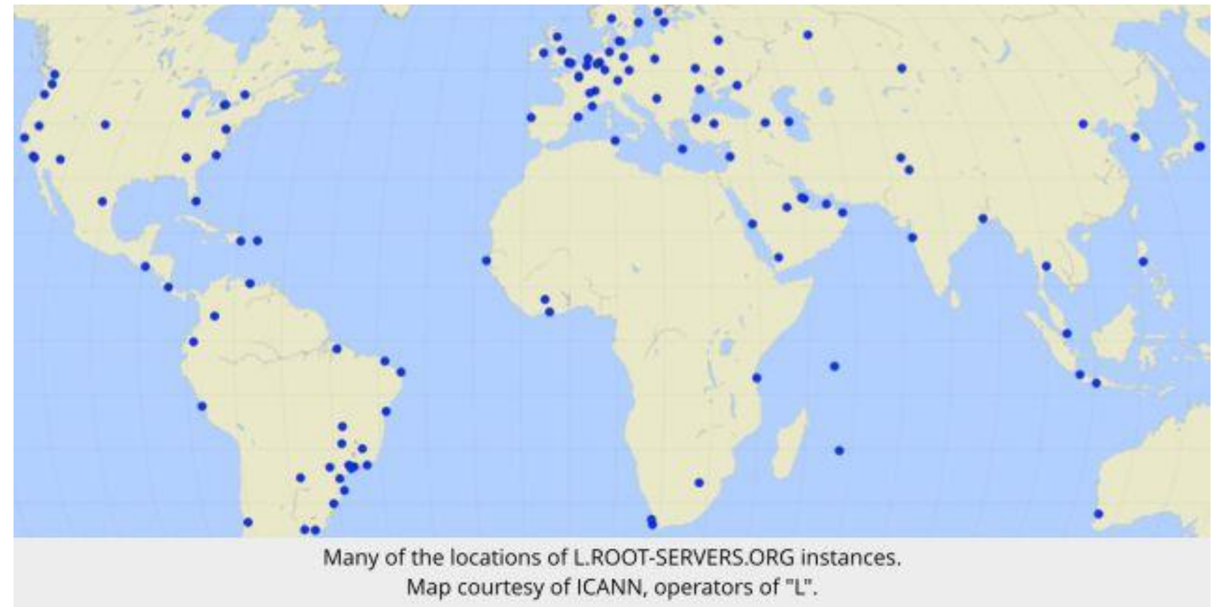
OST

Application Protocol: DNS

- l.root-servers.net , 1 root IP with anycast mirrored in 128 locations
  - All root servers
- 2015: Internet DNS servers withstand huge DDoS attack
  - 5m requests/s – some DNS could handle it
- Root zone is controlled by the United States Department of Commerce, operations by ICANN
- Root zone file: download

https://www.iana.org/domains/root/servers

HOSTNAME	IP ADDRESSES	MANAGER
a.root-servers.net	198.41.0.4, 2001:503:ba3e::2:30	VeriSign, Inc.
b.root-servers.net	199.9.14.201, 2001:500:200::b	University of Southern California (ISI)
c.root-servers.net	192.33.4.12, 2001:500:2::c	Cogent Communications
d.root-servers.net	199.7.91.13, 2001:500:2d::d	University of Maryland
e.root-servers.net	192.203.230.10, 2001:500:a8::e	NASA (Ames Research Center)
f.root-servers.net	192.5.5.241, 2001:500:2f::f	Internet Systems Consortium, Inc.
g.root-servers.net	192.112.36.4, 2001:500:12::d0d	US Department of Defense (NIC)
h.root-servers.net	198.97.190.53, 2001:500:1::53	US Army (Research Lab)
i.root-servers.net	192.36.148.17, 2001:7fe::53	Netnod
j.root-servers.net	192.58.128.30, 2001:503:c27::2:30	VeriSign, Inc.
k.root-servers.net	193.0.14.129, 2001:7fd::1	RIPE NCC
l.root-servers.net	199.7.83.42, 2001:500:9f::42	ICANN
m.root-servers.net	202.12.27.33, 2001:dc3::35	WIDE Project



Many of the locations of L.ROOT-SERVERS.ORG instances.
Map courtesy of ICANN, operators of "L".

OST

Application Protocol: DNS

- DNS structure
  - TTL defines the duration in seconds that the record may be cached by any resolver. “0” means no cache. Recommendation: > 1d
- Type of records
  - SOA - Start of Authority record: serial number and different caching times
  - NS - Name Server Record – sets the authoritative name server for this zone. 2 NS records – round robin! more sophisticated LB: split horizon
  - MX - name and relative preference of mail servers
  - A/AAAA - IPv4/IPv6 Address Record
  - TXT - arbitrary and unformatted text
  - PTR - opposite of A /AAAA

```
$TTL 3D
$ORIGIN tomp2p.net.
@ SOA ns.nope.ch. root.nope.ch. (2018030404 8H 2H 4W 3H)
                NS              ns.nope.ch.
                NS              ns.jos.li.
                MX      10      mail.nope.ch.
                A               188.40.119.115
                TXT             "v=spf1 mx -all“
www             A               188.40.119.115
bootstrap       A               188.40.119.115
$INCLUDE "/etc/opendkim/keys/mail.txt“
$INCLUDE "/etc/bind/dmarc.txt"
```

```
dig dsl.hsr.ch
dig -x 152.96.80.48
dig mx tomp2p.net
dig tomp2p.net @ns.nope.ch
```

- **Let’s add ost.tomp2p.net!**

OST

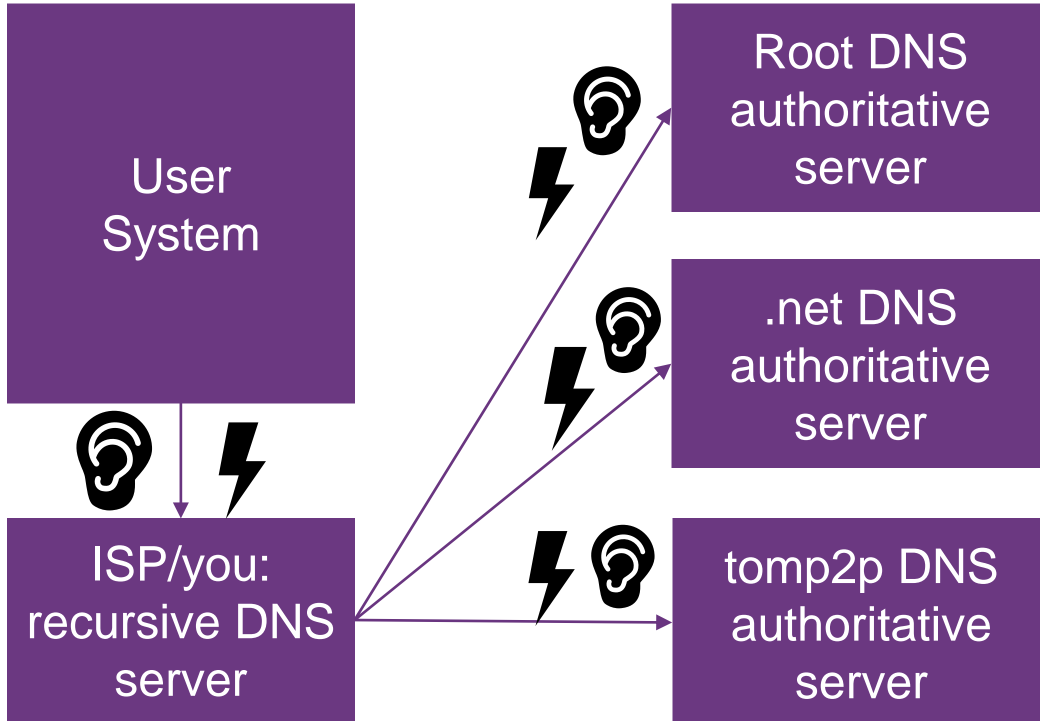
Application Protocol: DNS

- To run your own DynDNS service: TSIG
  - Enables DNS queries to authenticate updates to a DNS database
  - Uses shared secret and cryptographic hashing for authentication
- DNSSEC (security extension)
  - Authenticated and data integrity, **not** confidentiality
  - Can be used to bootstrap other security systems
    - Certificates, SSH fingerprints, IPSec pub keys
  - KSK: key signing keys to sign ZSK
  - ZSK: zone signing keys to sign records
    - Example: dig DNSKEY tomp2p.net
- New record types: RRSIG, DNSKEY, DS, …
  - RRSIG, sign all resource sets
  - DS (delegation signer) record in the parent zone
    - `dig DS tomp2p.net`
  - ZSK to sign RRset
    - How to validate ZSK?
  - KSK to sign ZSK pub key
    - With 2 keys, its easier to change ZSK
  - `dig any tomp2p.net`

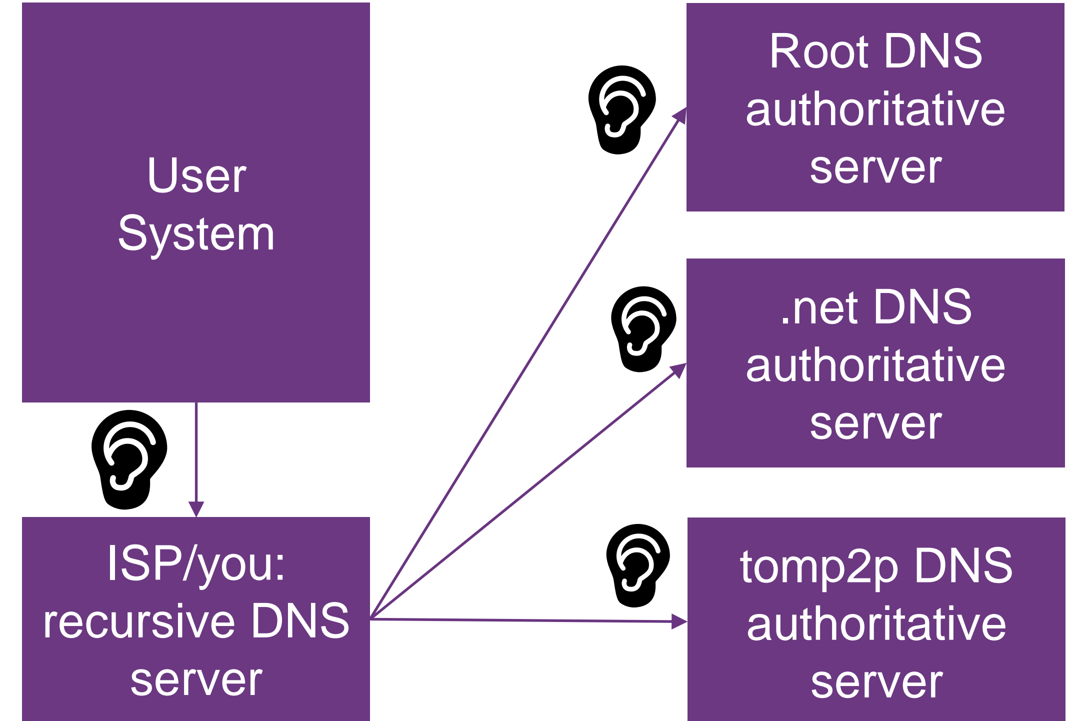
OST

Application Protocol: DNS

- DNS

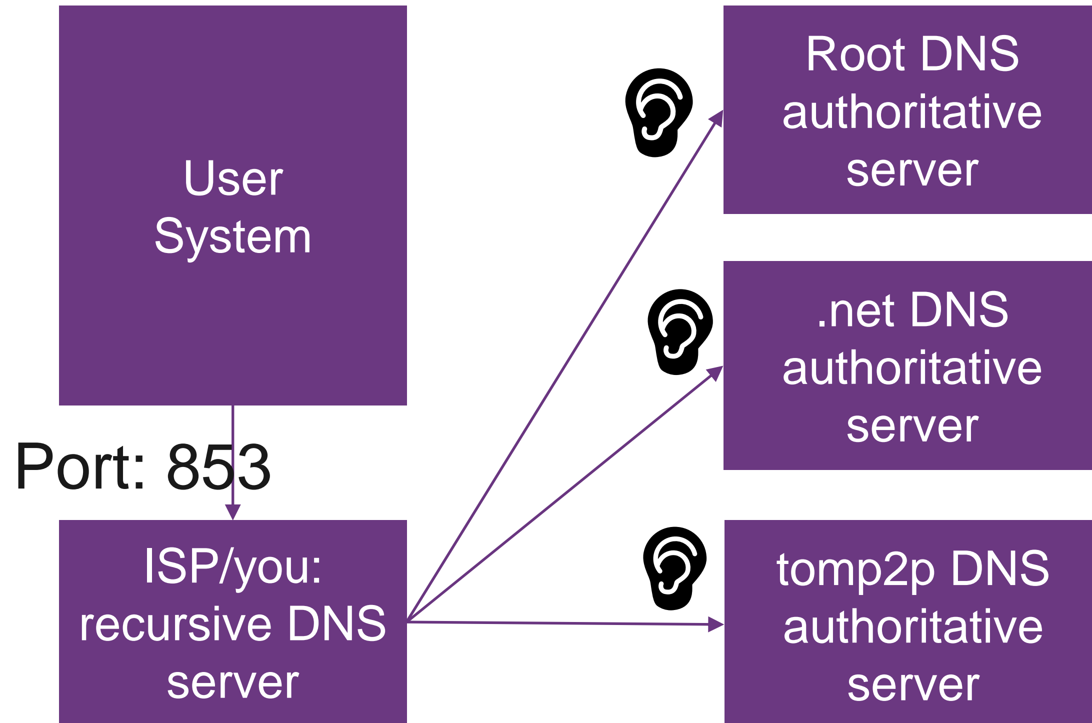


- DNSSEC



Application Protocol: DNS

- DoT (assuming DNSSEC)



- DoH (assuming DNSSEC)

User

System

Browser

ISP/you:
recursive DNS
server

Cloudflare:
recursive DNS
server

Root DNS
authoritative
server

.net DNS
authoritative
server

tomp2p DNS
authoritative
server

The DNS war

DoH

- provides confidentiality of lookups in transit
- Uses standard HTTP/2, on the standard port (443)
  - Cannot distinguish between traffic/DNS
- Trivially deployed , DNS response are served like simple web pages
- Performance: TCP+TLS handshake → 2/3 RTT
  - But: Cloudflare is close to you
- Difficult upgrade path for clients: per-application installation
- Browsers can perform DNS queries using Javascript

DoT

- provides confidentiality of lookups in transit
- DNS over TLS, separate port (853)
  - Can be blocked
- Widely supported by serving software (Bind, PowerDNS, Unbound) and public resolvers (Cloudflare, Quad9, Google)
- Performance: TCP+TLS handshake → 2/3 RTT
  - But: ISP is close to you
- Easy upgrade path for clients: clients can test if the configured resolver supports DoT on port 853, fall back to DoU53 otherwise)

OST

Let’s encrypt

- Non-profit CA
  - Provides certificates for TLS
  - Golive in 2016 (started in 2012), now issuing 2m certificates per day
- Certificates or domain-validation certificates. Cannot compete with traditional CA (identity checks)
- Certs are valid for 90 days, but automated renewal
  - ACME protocol – challenge response
    - Automated Certificate Management Environment
  - Query Web servers or DNS servers (wildcard)

- Certbot – client for ACME
  - `certbot certonly --webroot -w /tmp -d ost.tomp2p.net --debug-challenges`
  - Copy the challenge where Let’s encrypt server can find it (in my case /var/www/html)
  - Nginx config

```
server_name ost.tomp2p.net;
  ssl_certificate /etc/letsencrypt/live/ost.tomp2p.net/fullchain.pem;
  ssl_certificate_key /etc/letsencrypt/live/ost.tomp2p.net/privkey.pem;
```

  - This needs to be automated!

```
43 6 * * *    root    certbot renew --post-hook "systemctl reload nginx"
```

  - Caddy and Traefik already implement ACME