Protocol Buffers!

AKA: 0x 0a 11 50 72 6f 74 6f 63 6f 6c 20 42 75 66 66 65 72 73 21

Because **Binary Data Formats** are Exciting!

John Haldeman – Hackforge – November 2016

Seriously?

- Yeah!

- Binary Formats Can Help with Performance!
  - Protobuf is 3 to 10 times smaller than XML [1]
  - Protobuf is 20 to 100 times faster to parse than XML [1]

- But JSON and XML based protocols are easy to understand and use…
  - Goal: Make the messages small and translation fast – but still make it easy to use/understand

[1] https://developers.google.com/protocol-buffers/docs/overview#whynotxml

Protocol Buffers – Introduction

- Developed by Google – Used in many different projects there:

  At time of writing, there are 48,162 different message types defined in the Google code tree across 12,183 .proto files. They're used both in RPC systems and for persistent storage of data in a variety of storage systems. [2]

[2] https://developers.google.com/protocol-buffers/docs/overview#whynotxml

Protocol Buffers - Introduction

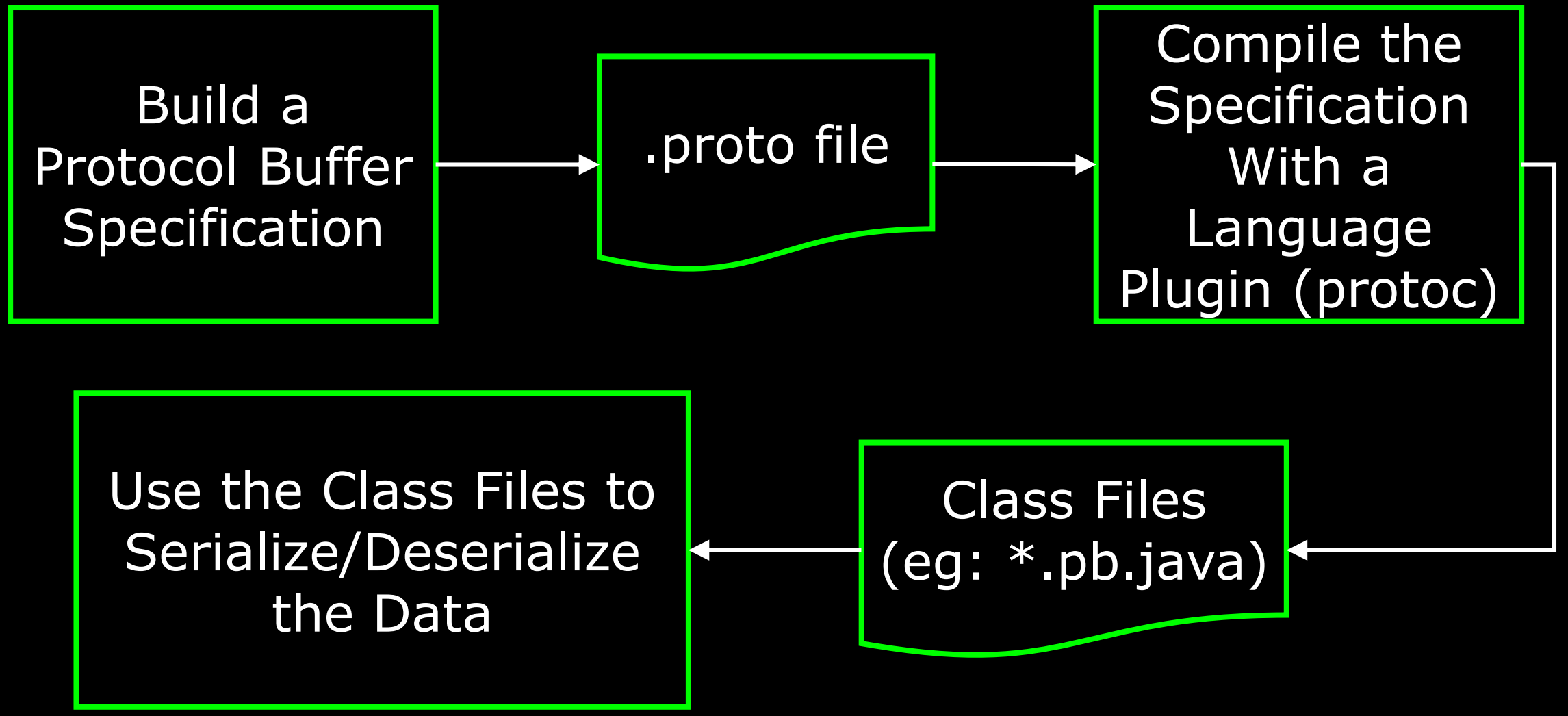
- What you get
  - A formal, cross-platform, cross-language definition of a message format
  - A set of tools to take the specification and build language specific APIs for serializing/deserializing the data [3]

[3] If you are unlucky enough that there are no current tools for your chosen language, you can build your own

Protocol Buffers – Introduction

- What you don’t get
  - A full wire protocol – You just get the data format, ***not*** the order of messaging, message delimiting, etc.
  - Although the libraries for each language may implement buffering – those are independent of protocol buffers themselves – which is language/library independent
- In other words you don’t get a protocol and you don’t really get buffers: Maybe it should actually be called: “Google Message Serializers”?

Protocol Buffer Process



Potential Drawbacks of Protocol Buffers

- You can’t read the data in a generic text editor
- While widely used, they are not as widely used as JSON/XML
- You rely on libraries/plugins that are probably less frequently maintained than JSON/XML parsing libraries (some of which are part of certain language’s standard libraries)
- Alternative binary formats (like Apache Thrift) have items like RPC specifications built in – Protocol Buffers requires supplementation (but those supplements are also Google support – like gRPC)
- Format not a community driven specification – evolves at Google’s direction
- Keeping track of message schemas requires stronger data governance than if the data was self-describing – if you don’t know the data format you can’t parse it

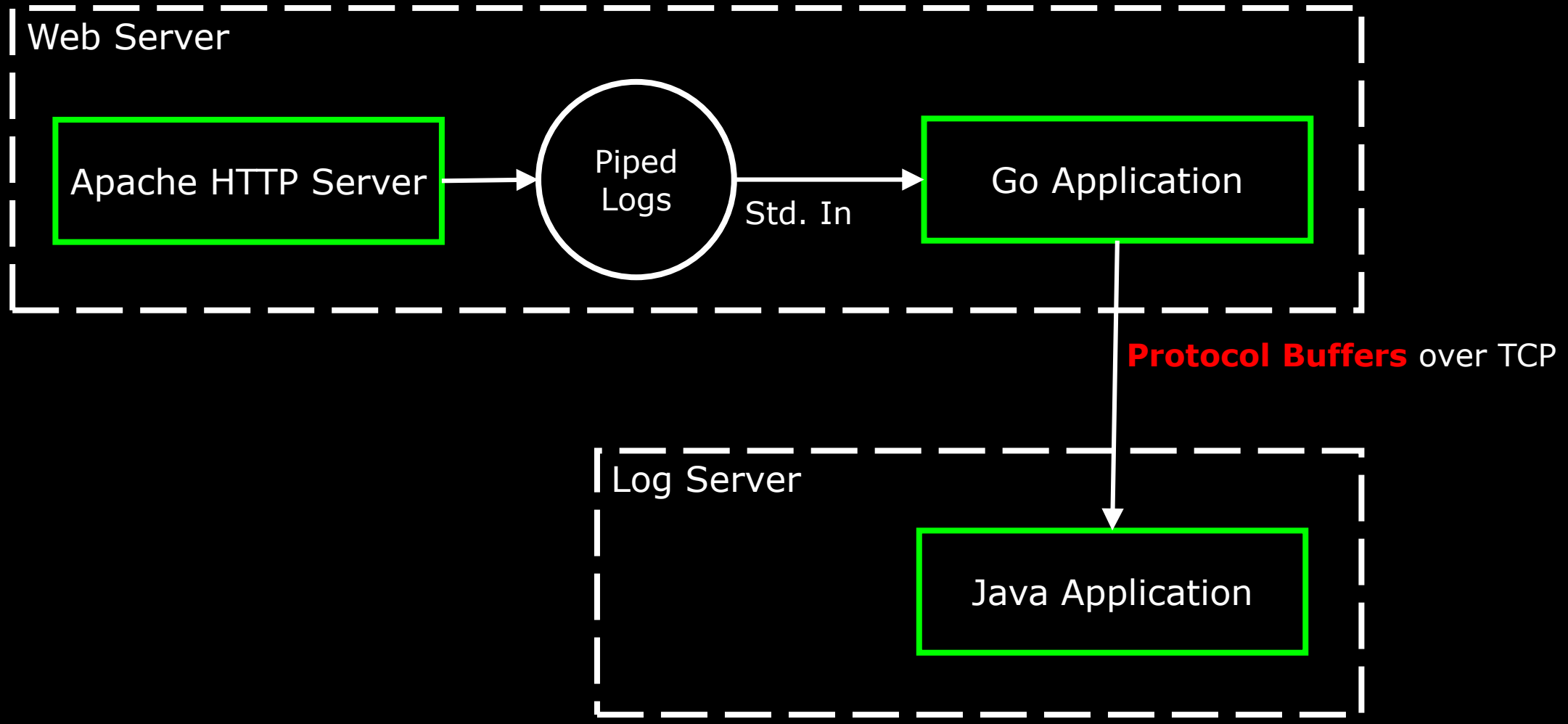
Who Knows What this is?

```
01 ORDERS.
   05 ORDER-ID          PIC X(5).
   05 CUSTOMER-ID       PIC X(5).
   05 ORDER-DATE.
      10 ORDER-YEAR     PIC 9(4).
      10 ORDER-MONTH    PIC 9(2).
      10 ORDER-DAY      PIC 9(2).
01 ITEM.
   05 PRODUCT-ID        PIC X(9).
   05 QUANTITY          PIC 9(4) BINARY.
   05 UNIT-COST         PIC 9(8) BINARY.
```

Do Something With Protocol Buffers

- Say you use Apache HTTP Server as either a front end web server or a PHP/CGI application server
- Say you want to monitor access to those servers, a lot of busy servers, remotely from a central server
- You want to do something with the data – send emails when suspicious stuff happens, aggregate data for reporting, monitor when request times start increasing, etc.

Architecture



1) Setting Up A Custom Log Message in Apache

In httpd.conf – Define a log format for the data we want:

LogFormat
"%{msec}t||%h||%v||%A||%H||%u||%m||%U||%q||%r||%>s||%X||%B||%D||%{Referer}i||%{User-agent}i||%{Accept}i||%{Accept-Language}i||%f" DelimDetails

%h - host
%u - HTTP Authenticated user
%{msec}t - number of milliseconds since the Epoch (1970-01-01 00:00:00)
%r - request line - GET /apache_pb.gif HTTP/1.0
%m - method (GET)
%U - URI (resource)
%q - query string
%H - protocol
%>s - status code
%B - size of content
%{Referer}i – Referer Request Header
%{User-agent}i - User Agent Request Header
%{Accept}i - Accept Request Header
%{Accept-Language}i – Accept-Language Request Header
%v - The canonical ServerName of the server serving the request.
%A - Local IP Address
%D - The time taken to serve the request, in microseconds
%f - filename
%X - Connection status when response is completed

2) Defining a Corresponding .proto File

We need to define a message format whose fields correspond with the log format we specified for apache.

monitorhttp.proto (2 columns shown below – 1 file):

```
package monitor.http;
message http_request{
        optional int64 timestamp = 1;
        optional string hostname = 2;
        optional string server_name = 3;
        optional string server_ip = 4;
        optional string protocol = 5;
        optional string http_user = 6;
        optional string method = 7;
        optional string resource = 8;
        optional string query = 9;
        optional string full_request = 10;
        optional int32 http_code = 11;
        optional string conn_status = 12;
        optional int64 content_size = 13;
        optional string time_to_serve = 14;
        optional string header_referer = 15;
        optional string header_user_agent = 16;
        optional string header_accept = 17;
        optional string header_accept_language = 18;
        optional string file = 19;
}
```

3) Compile the .proto File into Go Class Specifications

```
protoc --go_out=./ httpmonitorproto.proto
```

Generates: httpmonitorproto.pb.go

You can now use that Go API to interact with protocol buffer data

4.1) Go code: Initiate Connection, Create a Line Delimited Scanner on Standard Input (the rest of the code will go in [...])

```
func check(e error) {
    if e != nil {
        panic(e)
    }
}

func main() {
    conn, err := net.Dial("tcp", "10.10.9.1:8686")
    check(err)
    defer conn.Close()

    scanner := bufio.NewScanner(os.Stdin)
    for scanner.Scan(){
        [...]
    }
}
```

4.2) Go code: Inside the Scanner – When We Get a New Line, Split it and Parse Some Strings to Integers

```
splitLogLine := strings.Split(scanner.Text(), "||");
timeParse, err := strconv.ParseInt(splitLogLine[0], 10, 64)
check(err)
httpCodeParse, err := strconv.ParseInt(splitLogLine[10], 10, 32)
check(err)
contentSizeParse, err := strconv.ParseInt(splitLogLine[12], 10, 64)
check(err)
```

4.3) Go code: Inside the Scanner – Take the Split/Parsed Data and Throw it Into a new HTTPRequest message

```
test := &monitor_http.HttpRequest {
        Timestamp: proto.Int64(timeParse),
        Hostname:  proto.String(splitLogLine[1]),
        ServerName: proto.String(splitLogLine[2]),
        ServerIp: proto.String(splitLogLine[3]),
        Protocol: proto.String(splitLogLine[4]),
        HttpUser: proto.String(splitLogLine[5]),
        Method: proto.String(splitLogLine[6]),
        Resource: proto.String(splitLogLine[7]),
        Query: proto.String(splitLogLine[8]),
        FullRequest: proto.String(splitLogLine[9]),
        HttpCode: proto.Int32( int32(httpCodeParse) ),
        ConnStatus: proto.String(splitLogLine[11]),
        ContentSize: proto.Int64(contentSizeParse),
        TimeToServe: proto.String(splitLogLine[13]),
        HeaderReferer: proto.String(splitLogLine[14]),
        HeaderUserAgent: proto.String(splitLogLine[15]),
        HeaderAccept: proto.String(splitLogLine[16]),
        HeaderAcceptLanguage: proto.String(splitLogLine[17]),
        File: proto.String(splitLogLine[18]),
}
```

4.4) Go code: “Marshal” (ie: Serialize) the data into binary. Then write it to the socket

```
data, err := proto.Marshal(test)
check(err)

dataLength := int32(len(data))
binary.Write(conn, binary.BigEndian, dataLength)
conn.Write(data)
```

This code prefixes the protobuf message with a fixed-length 32-bit integer that tells the receiver how big the protobuf message is – Remember, you don’t get a wire protocol. This is a simple way to fill that gap

5) Piping Log Messages from Apache

In httpd.conf – Pipe the logs in the log format to our Go executable:

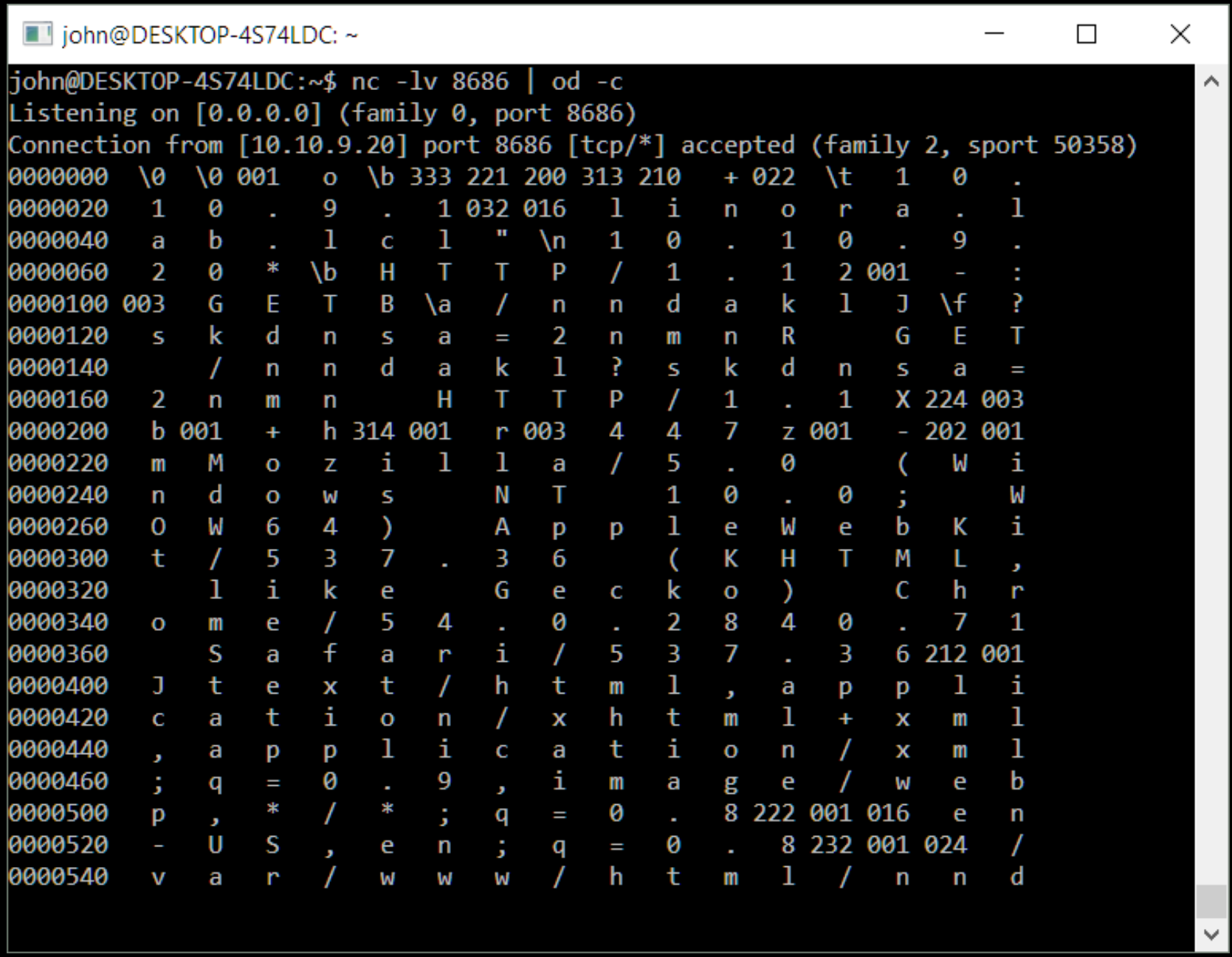
CustomLog "|httplogporotoforward" DelimDetails

“|httplogporotoforward” means: Run executable httplogporotoforward and pipe logs to std. input of that executable

Quick Sanity Check!

nc -lv 8686 | od -c

Start Apache and Generate a log message



6) Generate Some Java Class Files from the monitorhttp.proto file

```
protoc --java_out=./ httpmonitorproto.proto
```

Generates: monitor/proto/Monitorhttp.java

You can now use that Java API to interact with Protocol Buffer Data

7.1) Java Code: Wait for Data and Read a 32-bit Integer (the length of the protobuf message)

```
while(true){
    System.out.println("Accepted Connection");
    System.out.println("Waiting for Data");
    int ContentSize = inStream.readInt();
    if(ContentSize != 0){
        System.out.println("-------------------------------------");
        System.out.println("Wants to send " + ContentSize + " bytes of data");
        System.out.println("Reading...");
```

7.2) Java Code: Read the Protocol Buffer Data From the Socket

```
byte[] bytes = new byte[ContentSize];
inStream.read(bytes, 0, ContentSize);
http_request requestProto = http_request.parseFrom(bytes);
```

7.3) Java Code: Access The Data Using the Protocol Buffer APIs

```
System.out.println("Time of Request (milliseconds since epoch): " + requestProto.getTimestamp());
Instant inst = Instant.ofEpochMilli(requestProto.getTimestamp());
System.out.println("Time of Request (human readable): " + inst.toString());
System.out.println("This was the request string: " + requestProto.getFullRequest());
System.out.println();

Map<FieldDescriptor, Object> httpReqMap = requestProto.getAllFields();
Iterator<FieldDescriptor> keys = httpReqMap.keySet().iterator();
while(keys.hasNext()){
    FieldDescriptor protoField = keys.next();
    System.out.print(protoField.getName() + ": ");
    System.out.println(httpReqMap.get(protoField));
}
```

Protocol Buffers

- Faster, Smaller, Binary Representations of data
- With protoc compilation and APIs, you get some of the usability you'd expect from JSON/XML APIs (although none of the self-description)
- Since it's fairly widely used, libraries/plugins exist for most common languages – gives you language independence