Verteilte Systeme

Websockets (Quelle: WebSockets, Gorski et al, Hanser Verlag, 2015)

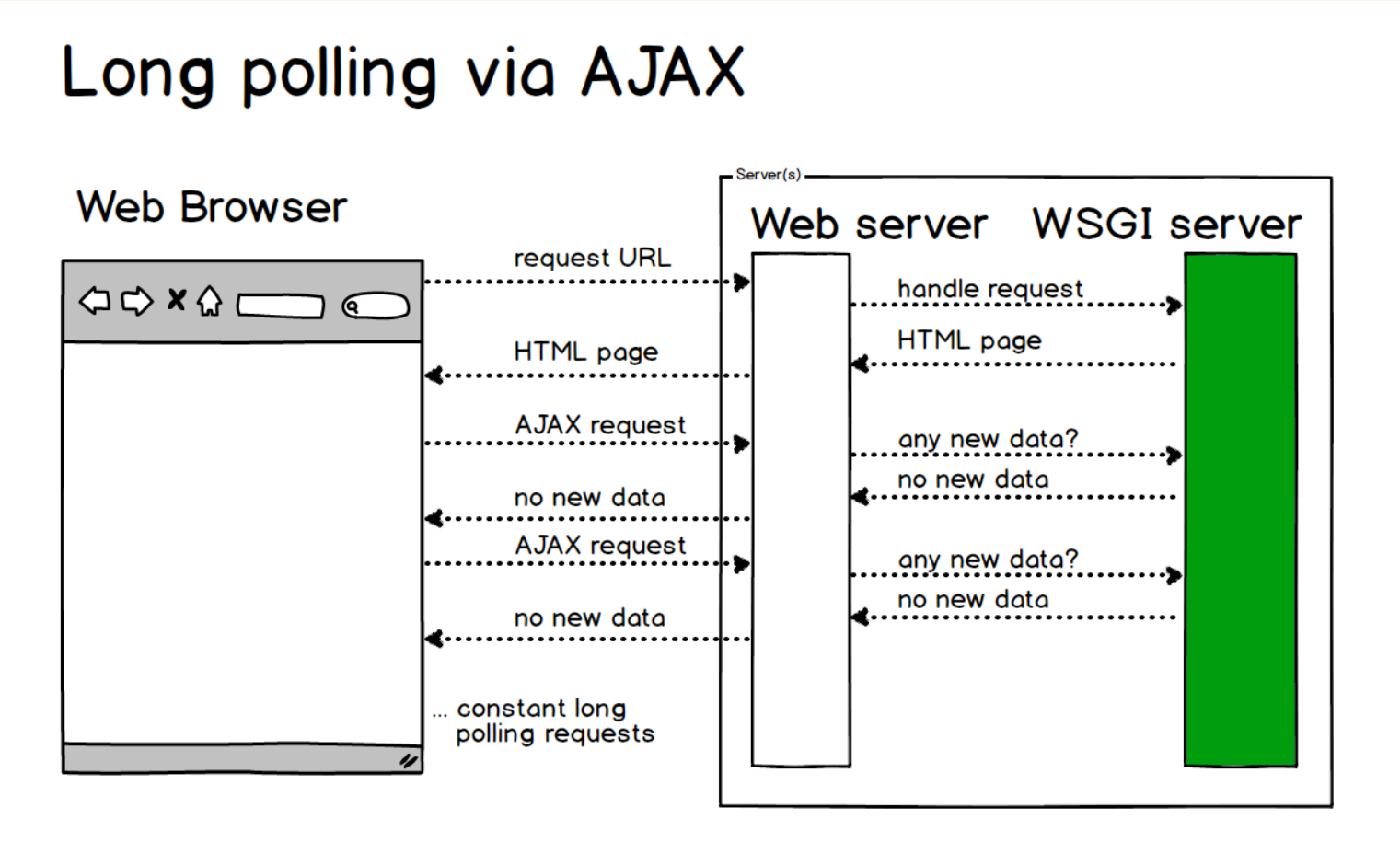
by

Dr. Günter Kolousek

HTTP/1.1

- Request/Response
  - $\rightarrow$ Interaktivitätsmöglichkeiten gering (half-duplex)
    - $\rightarrow$ keine Echtzeitfähigkeit
  - $\rightarrow$ kein spontanes Senden des Servers (d.h. kein Server-Push)
    - $\rightarrow$ kein Publish/Subscribe
- Header
  - $\rightarrow$ hoher Overhead

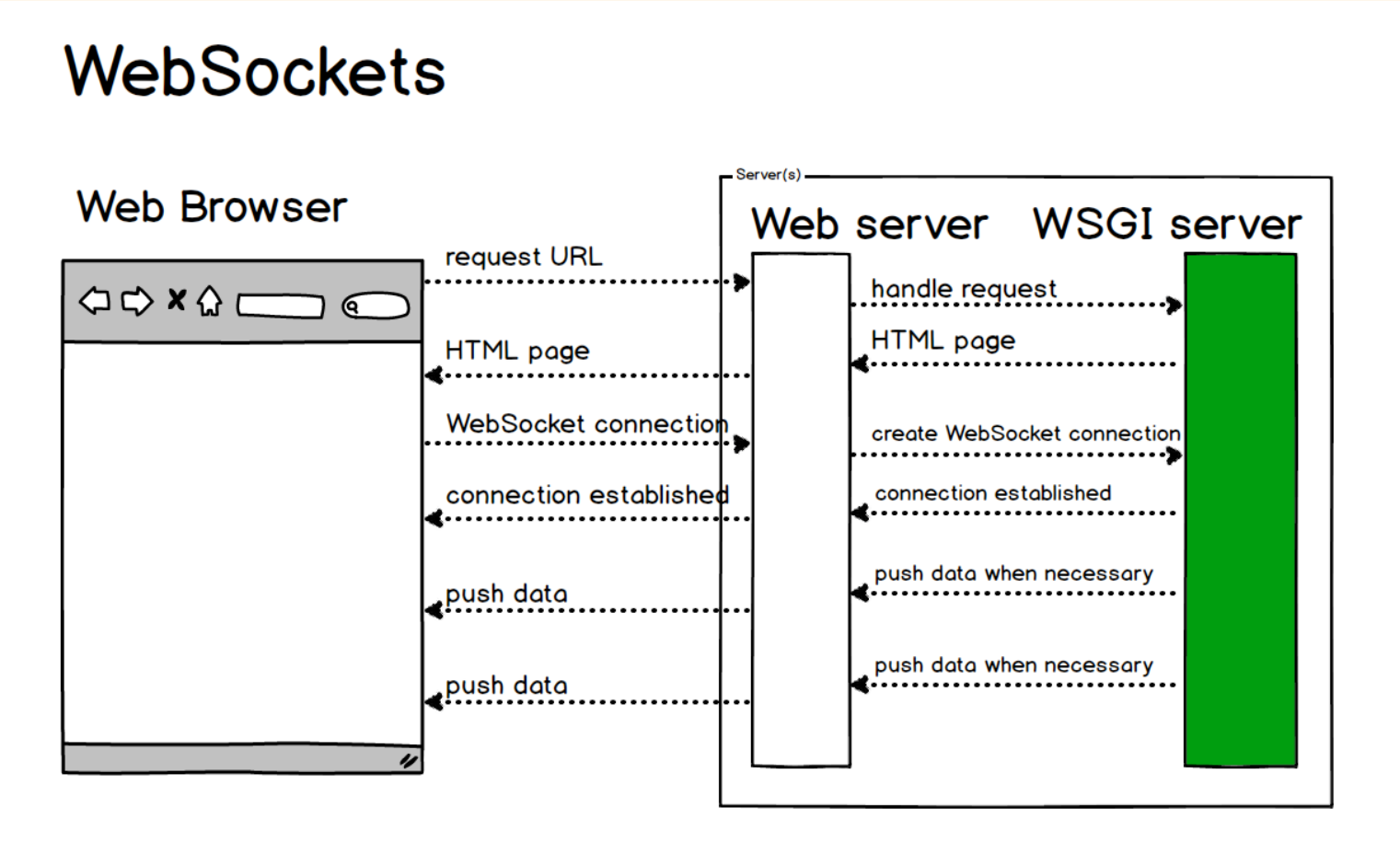
Long Polling



WebSockets…

- Vorteile
  - bi-direktional und full-duplex
    - anstatt half-duplex
  - Server-Push
    - anstatt polling|long polling|… (→ Request/Response)
  - geringer Overhead je Nachricht (anstatt Header…)
  - Port 80 bzw. 443 → keine Probleme mit Firewalls,…
- Nachteile?
  - kein Caching
    - kein Ziel!
  - nur tw. Unterstützung im IE
    - `json` als `responseType` fehlt
  - siehe Folie → Status quo

WebSockets... – 2



Anwendungsfälle

- schnelle Reaktionszeit
  - z.B. Chat-Applikation: Senden und gleichzeitiges Empfangen
- laufende Updates
  - z.B. Aktienkurse
- Ad-hoc Nachrichten
  - z.B. Nachrichtenversand (a la E-Mail)
- Viele Nachrichten mit geringer Größe
  - z.B. Watchdog

Protokoll

1. Handshake über HTTP
   1.1 Request
   1.2 Response
2. Datenübertragung
   - Frames (”Basic message framing”)
   - über TCP

Quelle: `https://tools.ietf.org/html/rfc6455`

Handshake – Request

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Origin: http://example.com
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
```

Handshake – Request – 2

- `Sec-WebSocket-Key`
  - Base64-kodierte Zeichenkette, die Zufallszahl enthält
  - dient zur Überprüfung, ob Server WebSockets unterstützt
- `Origin`
  - Herkunft, damit Server entscheiden kann, ob dieser annehmen will
  - wird vom Browser selbständig ausgefüllt
    - wirkt als Schutz gegen bösartiges JavaScript
  - **kein** Schutz vor beliebigen Clients!
- `Sec-WebSocket-Protocol` (optional)
  - Subprotokolle
- `Sec-WebSocket-Version`

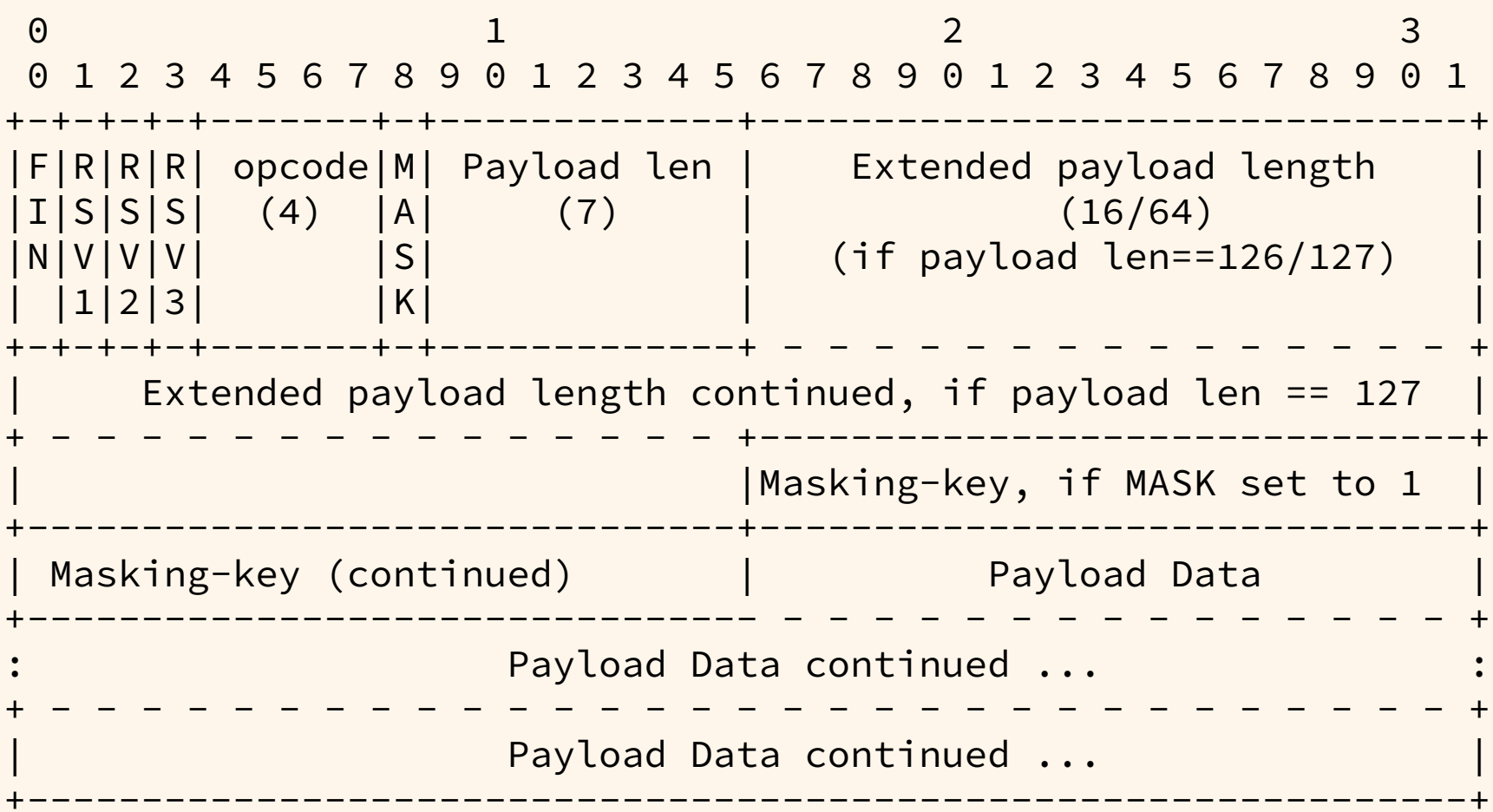
Handshake – Response

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
Sec-WebSocket-Protocol: chat
```

Handshake – Response – 2

- `Sec-WebSocket-Accept`
  - an `Sec-WebSocket-Key` wird ein GUID angehängt
    - festgelegt als: 258EAFA5-E914-47DA-95CA-C5AB0DC85B11
    - dann SHA-1
    - dann wieder Base64 kodiert
  - Client kann überprüfen

Datenübertragung – Frames



WebSockets-Frames – 2

- `FIN` ... $\rightarrow$ Fragmentierung
- `RSV1`, `RSV2`, `RSV3` ... reserviert
- `opcode`
  - Daten Frames (non-control frames): MSB $= 0$
    - `0x0` ... continuation frame (Fortsetzungsrahmen)
    - `0x1` ... text frame (gesamter Text muss UTF-8!)
    - `0x2` ... binary frame
    - `0x3` - `0x7` ... reserviert für weitere non-control frames
  - Steuer Frames (control frames): most significant bit $= 1$
    - `0x8` ... connection close
    - `0x9` ... ping frame
    - `0xA` ... pong frame
    - `0xB` - `0xF` ... reserviert für weitere control frames

WebSockets-Frames – 3

- `MASK` → Maskierung der Daten
- `Payload len`
  - 0-125 … aktuelle Länge der Daten; keine `Extended payload length` Felder im Header vorhanden
  - 126 … `Extended payload length` mit 2 Bytes
  - 127 … `Extended payload length` mit 8 Bytes
- `Extended payload length` entweder 0, 2 oder 8 Bytes je nach `Payload len`
- `Masking-key` → Maskierung der Daten
- `Payload data`
  - `Extension data` … optional, nur wenn eine Erweiterung ausverhandelt wurde
  - `Application data` … Länge: `Payload len` – Länge der `Extension data`

Fragmentierung

- Sinn und Zweck
  - Senden von Daten mit nicht bekannter Länge
  - Multiplexing
    - nur als Erweiterung zum WebSockets Protokoll
- Ablauf beim Senden von 3 Frames
  1. Frame: $\texttt{FIN} = 0, \texttt{opcode} \neq 0$
  2. Frame: $\texttt{FIN} = 0, \texttt{opcode} = 0$
  3. Frame: $\texttt{FIN} = 1, \texttt{opcode} = 0$

Maskieren der Daten

- MASK 1 $\rightarrow$ Payload wird mit `Masking-key` maskiert
  - muss bei Client-to-Server gesetzt sein
  - darf nicht bei Server-to-Client gesetzt sein
- `Masking-key` ... 0 oder 4 Bytes (je nach MASK); zufällige 32 Bit Zahl (je Frame!)
- Algorithmus im ausführbaren Pseudocode:

```
payload_data = [i for i in range(10)]
masking_key = [1, 2, 3, 4]
masked_data = []
for i, b in enumerate(payload_data):
    masked_data.append(b ^ masking_key[i % 4])
print(masked_data)
```

Ergebnis:

```
[1, 3, 1, 7, 5, 7, 5, 3, 9, 11]
```

Maskieren der Daten – 2

- Angriff auf transparente Proxies
  - Proxies, die WebSockets nicht **korrekt** unterstützen…
- Vorgang
  1. A erstellt WebSockets-Verbindung
  2. **In** den Daten folgt:

     ```
     GET /sensitive-doc HTTP/1.1
     Host: target.com
     ```
  3. Proxy interpretiert dies als Request und sendet diesen!
  4. Proxy empfängt Response und legt diesen in **Cache** ab
  5. **Irgendein** Benutzer greift auf `/sensitive-doc` von `target.com` zu und erhält falsche Version aus dem Cache!
- ”Abwehr”: Maskieren der Daten
  - Proxy erkennt diese nicht mehr

Control Frames

- keine Fragmentierung der Control Frames!
- Close
  - WebSockets-Verbindung schließen $\rightarrow$ senden von Close-Frame
  - Empfänger muss mit Close-Frame antworten (außer schon gesendet)
  - nach Senden von Close-Frame kein Senden von Daten mehr erlaubt
  - wenn Payload vorhanden
    - ersten zwei Bytes sind VZ-lose ganze Zahl mit Statuscode (in network byte order!): dzt. definiert 1000 bis 1011
    - danach kann: UTF-8 kodierter Text (für Grund)
  - danach kann TCP-Verbindung geschlossen werden
    - geht einer der Close-Frames verloren $\rightarrow$ Timeout

Control Frames – 2

- Ping
  - kann Payload enthalten
  - Zweck:
    - um Verbindung aufrecht zu halten ($\rightarrow$ Proxy)
    - um zu überprüfen, ob entfernter Endpunkt noch ”lebt”
- Pong
  - muss die selbe Payload enthalten wie Ping
  - kann unaufgefordert gesendet werden $\rightarrow$ Heartbeat in eine Richtung
    - darauf wird keine Antwort erwartet

API

- URLs für WebSockets
  - `ws:` unverschlüsselt
  - `wss:` verschlüsselt (mit TLS)
- Zustände
  - `CONNECTING` (`readyState` $= 0$)
  - `OPEN` (`readyState` $= 1$)
    - ab jetzt kann gesendet werden
  - `CLOSING` (`readyState` $= 2$)
  - `CLOSED` (`readyState` $= 3$)
- Konstruktor `WebSocket(url[, protocols])`
- Event-Handler
  - `onopen`, `onmessage`, `onclose`, `onerror`

Beispiel

```
var ws = new WebSocket("ws://echo.websocket.org")

ws.onopen = function() {
    console.log("open"); ws.send("hallo");
}
ws.onmessage = function(message) {
    console.log(message.data); ws.close();
}
ws.onclose = function(event) {
    console.log("closed...");
}
ws.onerror = function(event) {
    console.log("Fehler: " + event.reason +
                "(" + event.code + ")");
}
```

`WebSocket` – Attribute

- `binaryType` … String: entweder ”Blob” oder ”ArrayBuffer”
  - `send(Blob data)`, `send(ArrayBuffer data)`
- `bufferedAmount` … long: Anzahl der Bytes, die noch in Queue und noch nicht versendet (read-only)
- `extensions` … String: ausgehandelte Extensions (read-only)
- `protocol` … String: aktuelles Subprotokoll (read-only)
- `url` … String: URL (read-only)

Status quo

- Probleme
  - Implementierungen (Browser, Server) fehlerhaft
  - Proxies: fehlerhaft bzw. keine WebSockets-Unterstützung!
  - Autorisierung: kein Zugriff auf Header über JS API
- Richtlinien
  - immer TLS verwenden
    - $\rightarrow$ Sicherheit, Proxies!
  - one-time-token zur Autorisierung verwenden
    - Request an Server $\rightarrow$ generiert Token mit timeout $\rightarrow$ legt es am Server ab $\rightarrow$ Token wird zurückgeschickt $\rightarrow$ WebSockets Verbindung öffnen $\rightarrow$ Token senden
  - einen eigenen Server für WebSockets verwenden
  - eingehende Daten immer validieren (Client & Server)

Quellen: RFC6455, `http://lucumr.pocoo.org/2012/9/24/websockets-101/`