



Berner Fachhochschule
Technik und Informatik

Client/Server, Sockets

Kernmodul Embedded Systems

V 1.0 © 2013 by roger.weber@bfh.ch

Lernziele

Die Kursteilnehmer sind in der Lage:

- Einsatzgebiete für Embedded Webserver zu erkennen.
- Den Aufbau einer TCP oder UDP-basierten Socket-Verbindung zu erklären.
- Eine Webserver-Applikation basierend auf Sockets zu implementieren.

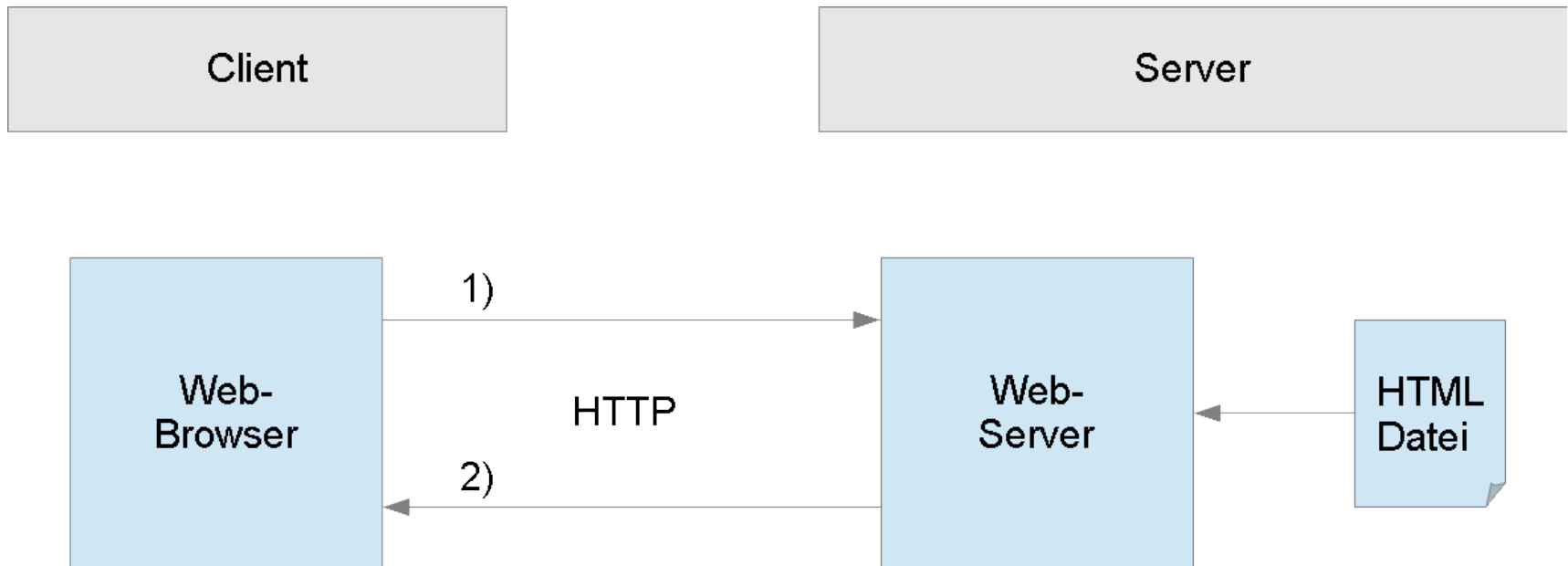
Inhalt

- Client/Server Kommunikation
 - Statische und dynamische Webseiten
 - Programmiersprachen für Internet-Anwendungen
 - Embedded Webserver
- Socket-Programmierung
 - Repetition Socket
 - TCP und UDP-basierter Kommunikationsaufbau
 - Socket-Programmierung in C
 - WebSocket-Programmierung in JavaScript

Statische Webseiten

- Information ist in statischer Form auf dem Server abgelegt.
- Technologien wie:
 - HTML (Auszeichnungssprachen)
 - CSS (Layoutsprachen)
 - Plug-Ins

Darstellung statischer Webseiten

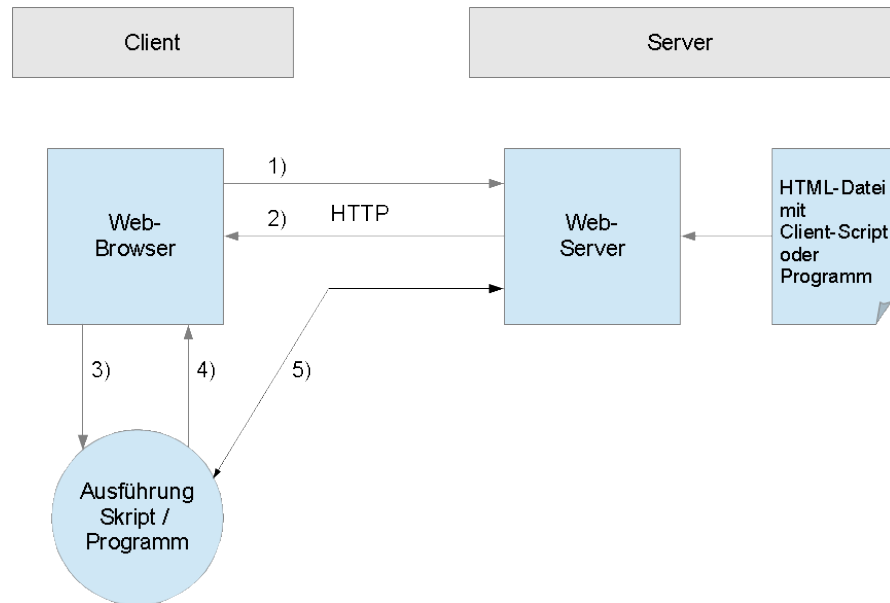


- 1) Der Browser fordert eine Webseite vom Server an.
- 2) Der Server liefert die Webseite.

Dynamische Webseiten

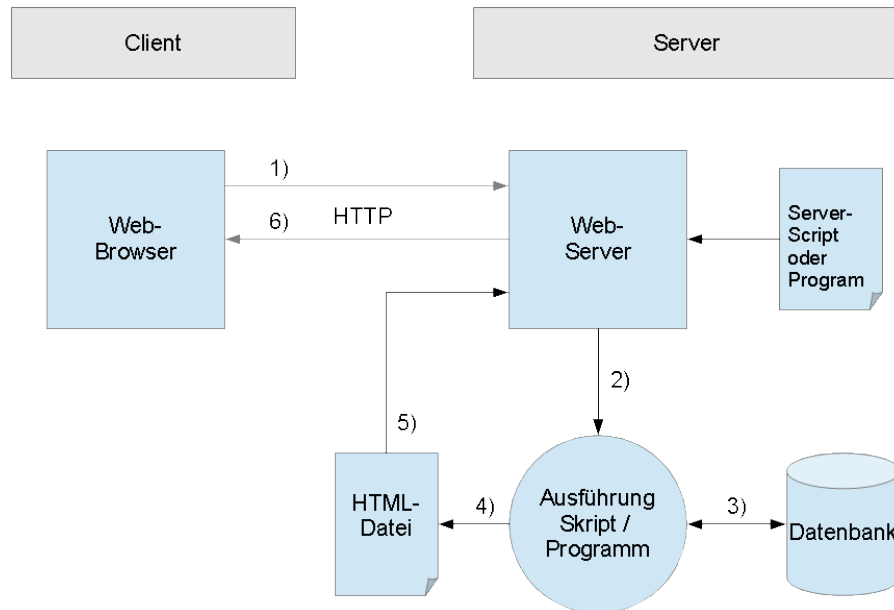
- Serverseitig dynamische Webseiten:
 - Werden auf dem Server zur Laufzeit erstellt.
- Clientseitig dynamische Webseiten :
 - Kontextabhängige Menüs
 - Eingabeprüfung
 - Berechnungen
 - Kommunikation (z.B. WebSockets)
 - Canvas

Clientseitige dynamische Ausführung



- 1) Browser fordert vom Server Webseite an
- 2) Server liefert Webseite inkl. Skript / Programm
- 3) Client stellt Webseite dar, startet Skript / Programm
- 4) Dynamische Darstellung der Daten auf dem Client
- 5) Kommunikation mit dem Server

Serverseitige dynamische Ausführung



- 1) Der Browser fordert vom Server eine Webseite an.
- 2) Server startet Skript / Programm
- 3) Skript / Programm liest Daten aus DB
- 4) Skript / Programm erstellt anwenderspezifische Webseite
- 5) Skript übergibt die Webseite dem Webserver
- 6) Client erhält die anwenderspezifische Webseite

Vergleich der Skript- und Hochsprachen

	Skriptsprachen	Hochsprachen
Aufgabe	Client: kleine Hilfsprogramme, Plugins Server: Erzeugung dynamischer Webseiten	
Ausführung	Skript wird durch Interpreter ausgeführt	Compiler erzeugt Bytecode oder Objektcode
Einarbeitungsaufwand	niedrig bis mittel	mittel bis hoch
Entwicklungsumgebung	Texteditor oder Script-Tools	Entwicklungsumgebungen , z.B. Eclipse für Java-Applets oder Visual Studio für SW-Komponenten und Programme.
Einbindung in HTML-Seite	Direkt in der HTML-Seite integriert (JavaScript) .	HTML-Seite enthält Verweis auf Plugins. Falls diese noch nicht vorhanden sind, werden sie geladen.
Sicherheit	Client: Beschränkter Zugriff auf Festplatte, Speicher, LAN für JavaScript . Voller Zugriff bei VBScript . Server: Voller Zugriff auf Datenbanken, I/O-System usw. (z.B. Perl)	Client: Kein direkter Zugriff auf Festplatte, Speicher und LAN für Java Applets . Erweiterter Zugriff für SW-Komponenten . Server: Voller Zugriff auf Datenbanken, I/O-System usw.

Übersicht der Skript-und Hochsprachen

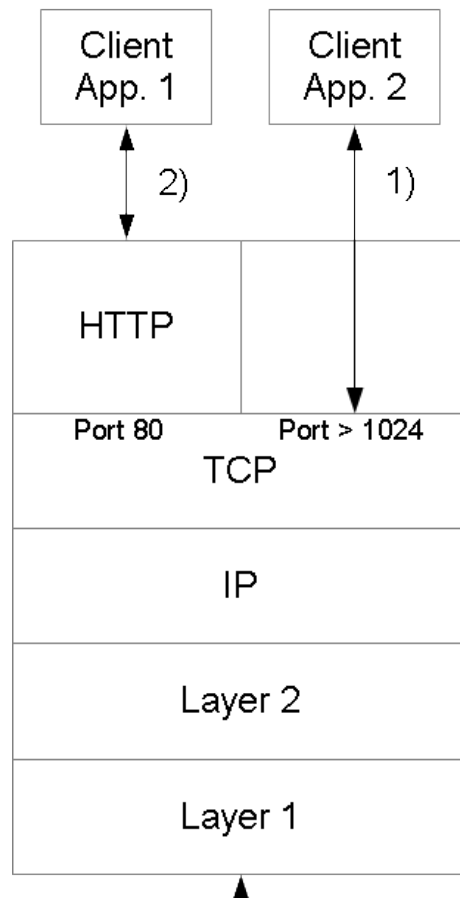
	Sun	Standard	Microsoft
Client, statisch		HTML, CSS XML	
Client, dynamisch		JavaScript Java Applet	VBScript Jscript .net
Server	JSP Java Servlet	CGI / Perl PHP Kompillierte Prog.	ASP .net

Embedded Webserver

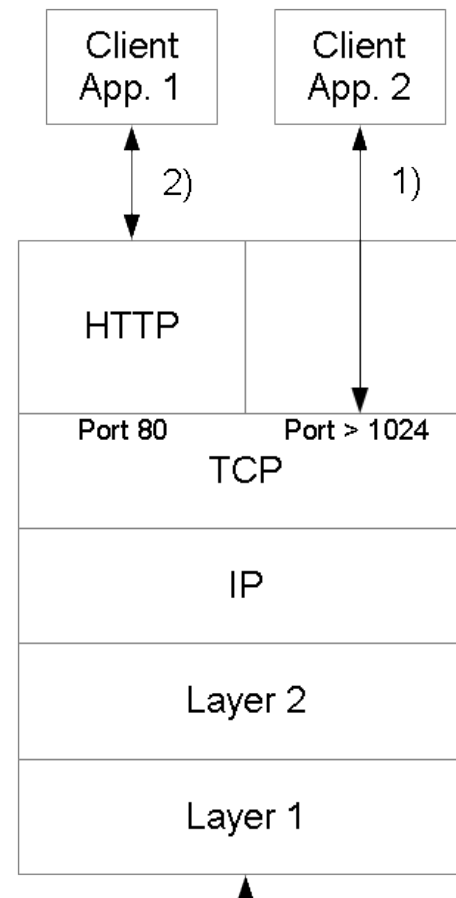
- Einsatz in Embedded Systemen
→ Maschine, Automaten, Anlagen...
- Informationen eines technischen Prozesses auf dem Client visualisiert.
- Möglichkeit für:
 - Fernwartung (Diagnose, Konfiguration, Parametrisierung)
 - Alarmierung

ISO/OSI-Modell für Embedded Webserver

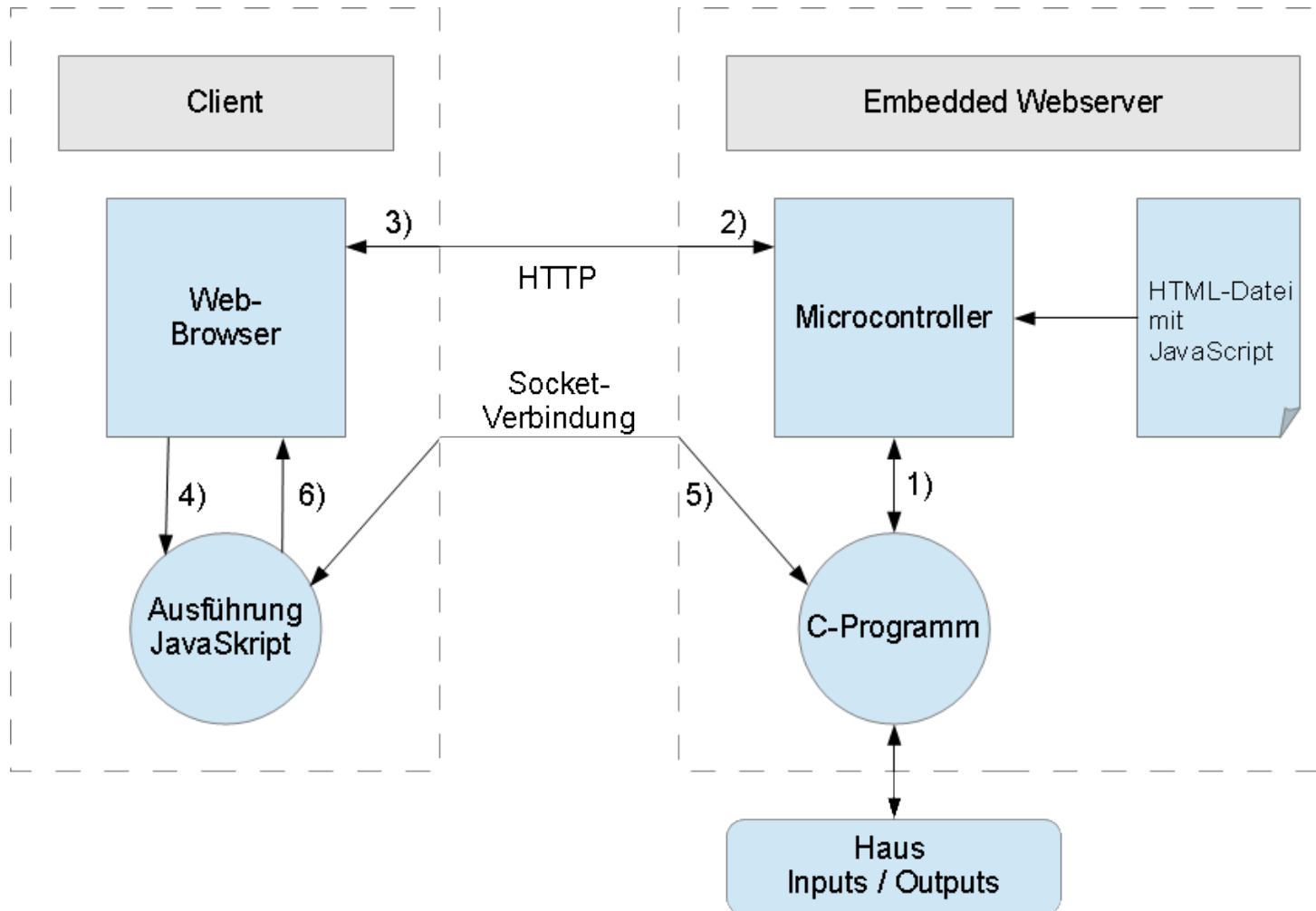
Client / Browser



Embedded Webserver



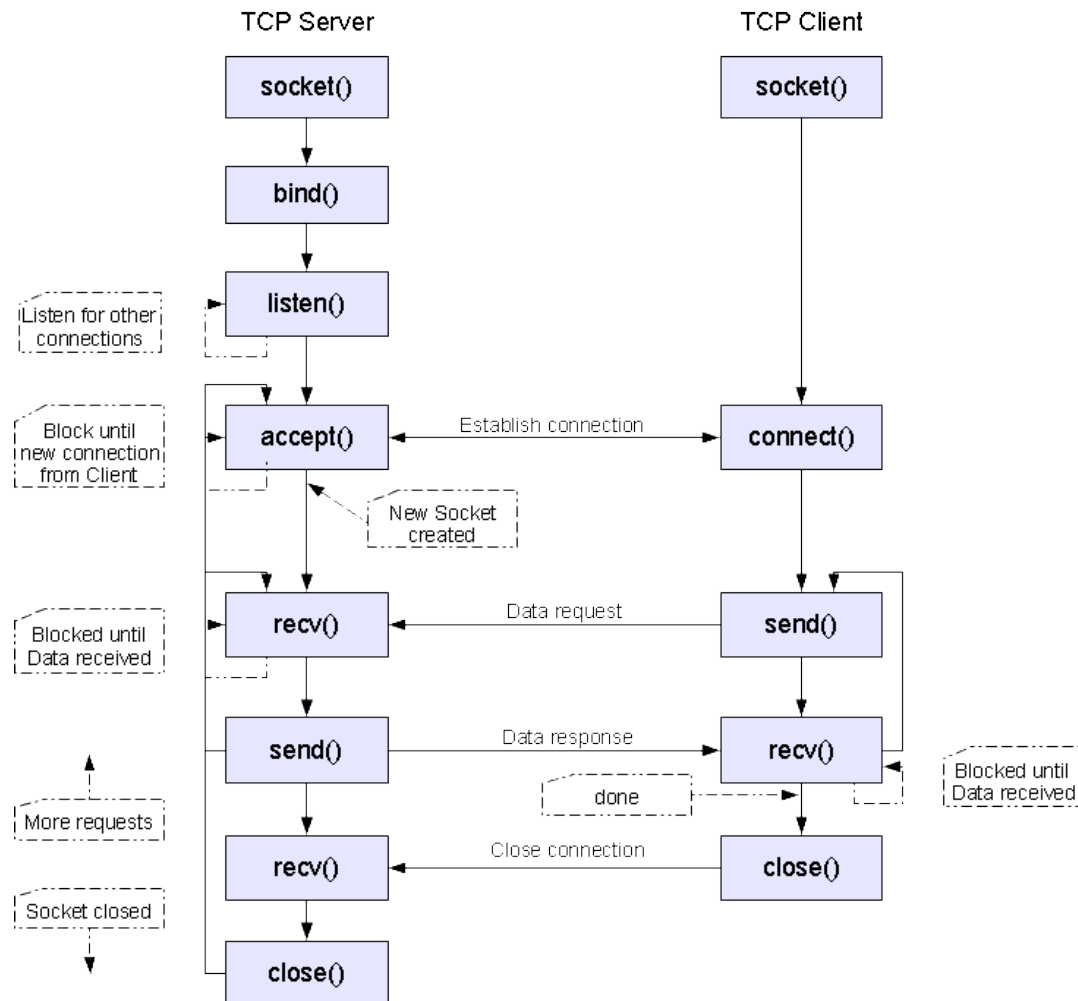
Ablauf einer Client-Server Kommunikation



Eigenschaften Socket

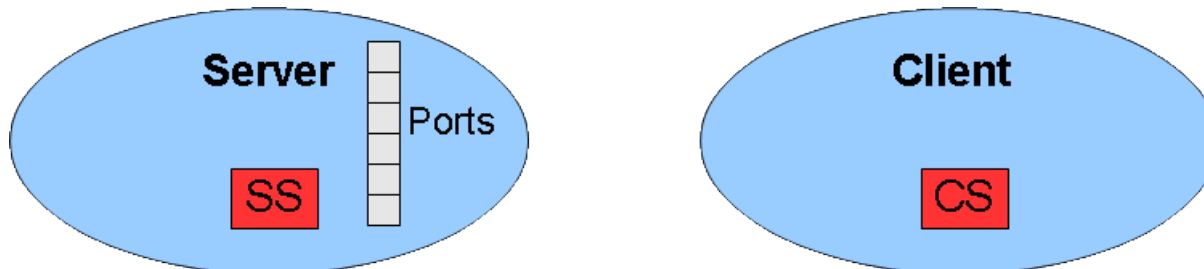
- Ein **Socket** ist ein eindeutig bestimmter Endpunkt einer Kommunikation. Er wird beim Internet-Protokoll durch folgendes Tripel definiert:
 - IP-Adresse
 - Protokoll (z.B. TCP)
 - Port-Nummer
- Sockets werden für die Kommunikation zwischen zwei Applikationen auf zwei Rechnern verwendet.
- Generell ist folgendes Vorgehen einzuhalten, um Daten zwischen zwei Rechnern auszutauschen:
 - Aufbau einer Verbindung über Sockets
 - Austausch der Daten
 - Abbau der Verbindung

TCP-basierter Kommunikationsaufbau



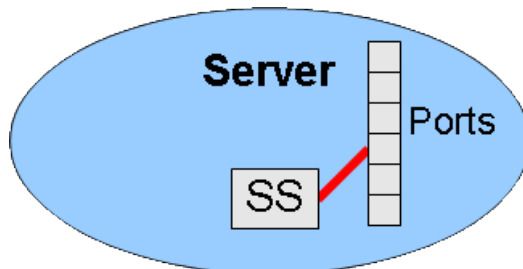
socket()

- Generiert einen TCP oder UDP Socket.
- Beim Server wird ein Server-Socket (SS), beim Client ein Client-Socket (CS) erzeugt.
- Bei der Erzeugung des Sockets muss das zu verwendende Protokoll (TCP,UDP) angegeben werden.



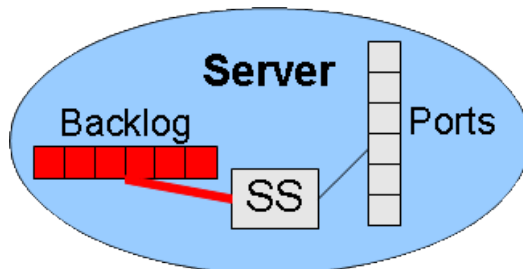
bind()

- Verbindet einen gegebenen Socket mit einer lokalen Protokoll-Adresse (IP- und Portnummer).
- Wird nur vom Server verwendet.



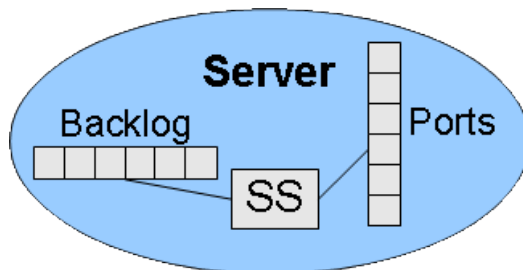
listen()

- Setzt den Server in einen passiven Lauschmodus.
- Der Aufruf wird von verbindungsorientierten Servern (TCP) verwendet und signalisiert die Empfangsbereitschaft.
- Backlog gibt die Anzahl der möglichen Verbindungsanforderungen an, die max. in die Warteschlange gestellt werden können.



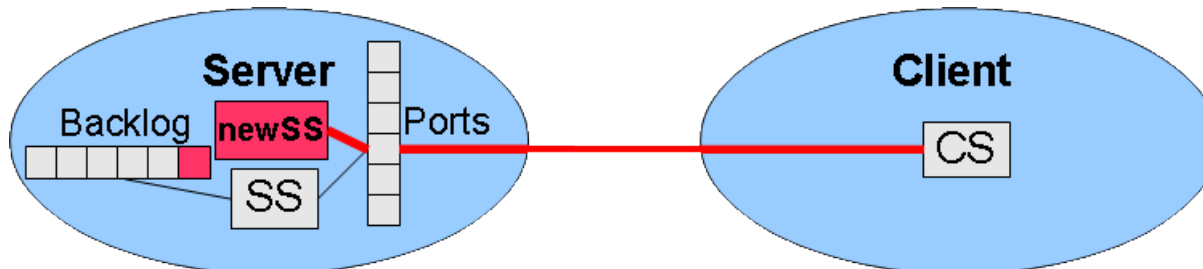
accept()

- Der Server wartet auf einen Verbindungswunsch des Clients bzw. nimmt den nächsten Verbindungswunsch aus der Warteschlange (Backlog).
- Falls die Warteschlange leer ist, wird solange gewartet, bis ein neuer Client eine Verbindung verlangt.
→ **Blockierend!**
- Je nach Implementation des TCP/IP Stacks wird auch nicht blockiert.



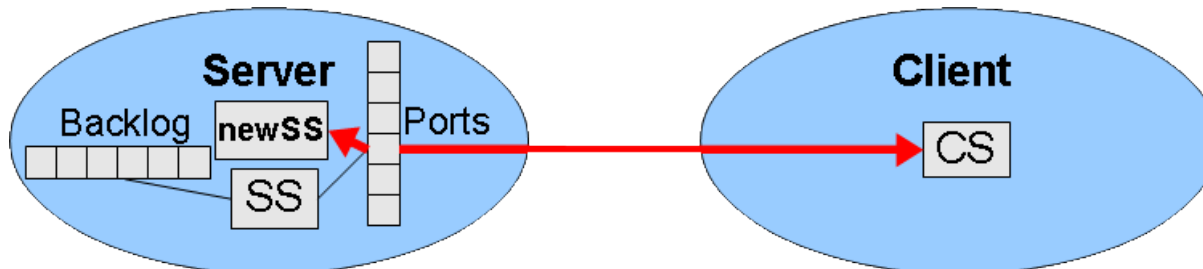
connect()

- Der Client verlangt mit connect() beim Server eine Verbindung.
- Der Server-Socket kreiert einen neuen Socket (newSS), der von nun an eine Art Standleitung zum anfragenden Client unterhält.
- Der Server-Socket (SS) ist so nach wie vor bereit, von anderen Clients über den selben Port Verbindungsanfragen entgegenzunehmen.



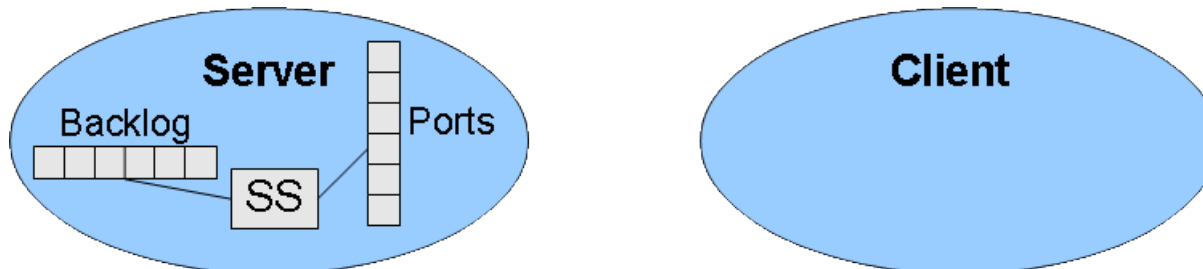
send() / recv()

- Nun können zwischen Client und Server mit `recv()` und `send()` bidirektional Daten ausgetauscht werden.
- Das Empfangen von Daten durch `recv()` kann blockierend oder nicht-blockierend durchgeführt werden.

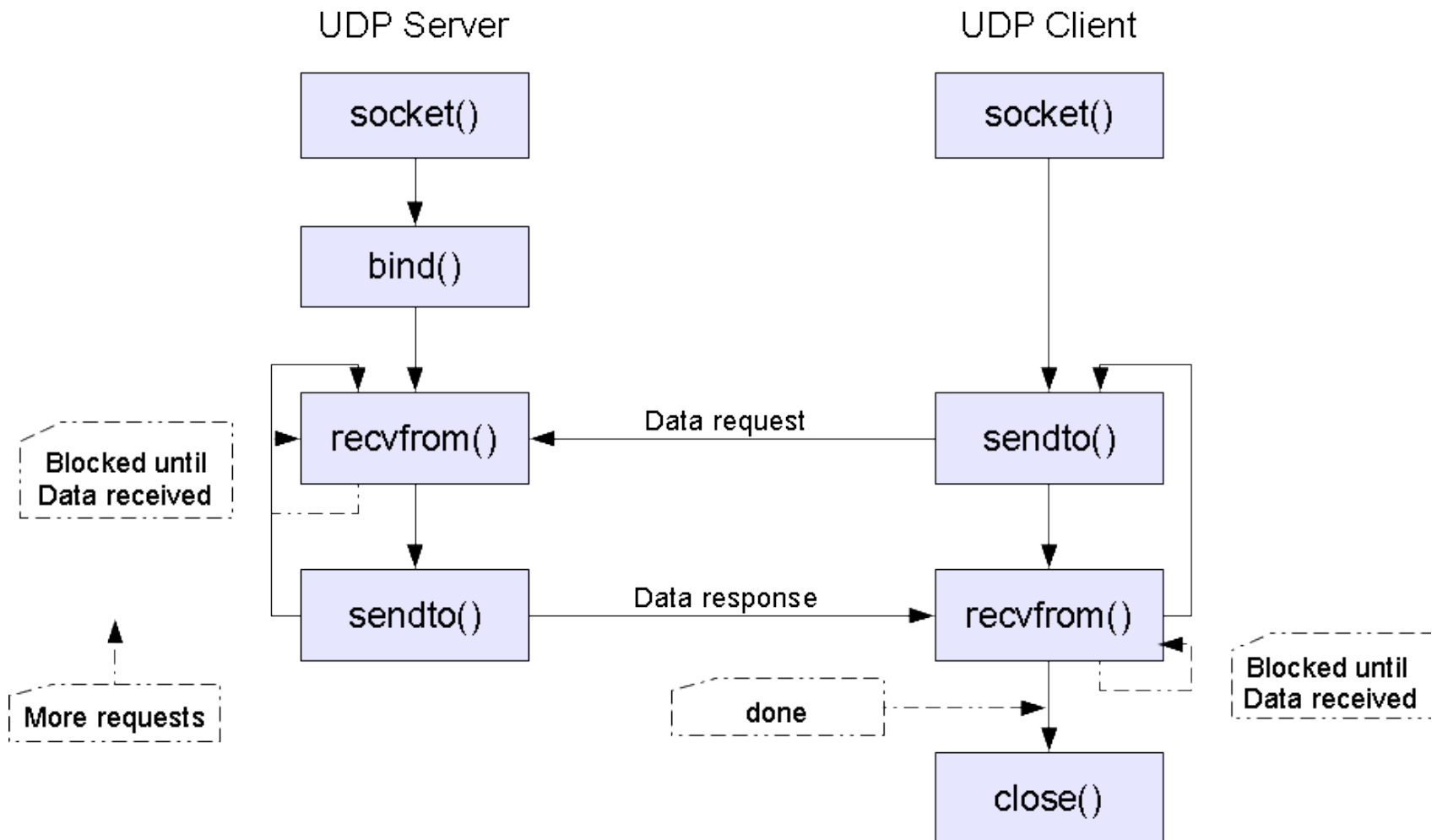


close()

- Schliessen der Sockets:
 - auf dem Server newSS
 - auf dem Client CS



UDP-basierter Kommunikationsaufbau



Datenstrukturen und Hilfsfunktionen

- **Struktur für IP-Adressen**

```
struct in_addr{  
    in_addr_t s_addr; /* IP address (32 bits ) */  
};
```

- **Protokolladresse für die Adressfamilie AF_INET**

```
struct sockaddr_in{  
    uint8_t sa_len; /* structure size */  
    sa_family_t sin_family; /* Address Family ( AF_INET ) */  
    in_port_t sin_port; /* Address port (16 bits ) */  
    struct in_addr sin_addr; /* IP address (32 bits ) */  
    char sin_zero[8]; /* Not used */  
};
```

- **Network Byte Order (Big Endian)**

```
# include <arpa/inet.h>  
uint32_t htonl(uint32_t hostlong);  
uint16_t htons(uint16_t hostshort);
```


socket()

- Prototyp

```
int socket(int domain, int type, int protocol);
```

- domain: Protokoll-Familie, für IPV4: AF_INET
- type: Typ des Sockets beziehungsweise sein Verhalten
- protocol: Definition des Protokolls

- Beispiel

```
tcpSock_id = socket(AF_INET, SOCK_STREAM,  
                    IPPROTO_TCP);
```

bind()

- Prototyp

```
int bind(int sock_id, const struct sockaddr * address,
         socklen_t address_len );
```

- sock_id: Socket-ID, welche von socket() zurückgegeben wurde.
- addr: Pointer auf struct sockaddr. Protokoll-Adresse (Port, IP).
- addrlen: Grösse der übergebenen Datenstruktur sockaddr.

- Beispiel

```
struct sockaddr_in address ;
address.sin_family = AF_INET ;
address.sin_port = htons( SERVER_PORT_NBR );
address.sin_addr.s_addr = htonl( INADDR_ANY );
```

```
bind_status = bind(sock_id, (struct sockaddr *)&address,
                   sizeof(struct sockaddr_in ));
```

listen()

- Prototyp

```
int listen (int sock_id, int backlog );
```

- sock_id: Socket-ID, welche von socket() zurückgegeben wird.
- Backlog: Gibt die Länge der Warteschlange für Clients an, die für diesen Socket verwaltet werden können.

- Beispiel

```
const int BACKLOG = 5;  
listen_status = listen(sock_id, BACKLOG);
```

accept()

- Prototyp

```
int accept(int sock_id,  
          struct sockaddr *addr_remote,  
          socklen_t *addrlen_remote);
```

- sock_id: Socket-ID, welche von socket() zurückgegeben wird.
- addr_remote: Pointer auf struct sockaddr; die Funktion accept() wird Informationen über den Client in die Struktur abfüllen.
- addrlen_remote: Tatsächliche Länge der Struktur sockaddr.

- Beispiel

```
struct sockaddr_in addr_remote;  
socklen_t addrlen_remote;  
newSock_id = accept(sock_id,  
                   (struct sockaddr*)&addr_remote,  
                   &addrlen_remote);
```

new socket

1) Blocking
2) not blocking !

recv()

- Prototyp

```
ssize_t recv(int newSock_id, void *buf,
             size_t buf_len, int flags);
```

- newSock_id: Socket-ID von accept ()
- buf: Pointer auf Buffer für empfangene Daten
- buf_len: Länge des Buffers
- flags: Spezifiziert die Empfangs-Optionen
- Rückgabewert: Anzahl der gelesenen Bytes

- Beispiel

```
ssize_t rx_data_len;
char rxBuf[RX_BUFSIZE];
rx_data_len = recv(newSock_id, rxBuf,
                  RX_BUFSIZE, 0);
```

send()

- Prototyp

```
int send(int newSock_id, const void *data,
        size_t datalen, int flags);
```

- newSock_id: Socket-ID von accept ()
- data: Pointer auf die zu versendenden Daten
- datalen: Länge der zu versendenden Daten
- flags: Spezifiziert die Sende-Optionen
- Rückgabewert: >0 Anzahl erfolgreich gesendeter Bytes

- Beispiel

```
int tx_msg_len;
char txBuf[TX_BUFSIZE];
tx_msg_len = send(newSock_id, txBuf,
                  TX_BUFSIZE, 0);
```

close()

- Prototyp

```
int close(int sock_id) ;
```

- sock_id: Socket-ID, welche von socket() oder accept() beim erzeugen des Sockets zurückgegeben wurde.

- Beispiel

```
close(newSock_id) ;
```

Erzeugen eines WebSockets in JavaScript

- Prototyp

WebSocket (URL) ;

- URL: URL des WebSocket-Servers
- Format:Protokoll (ws://), URL WebSocket-Server, Sub-Protokoll

- Beispiel

```
var exampleWebSocket = new
    WebSocket('ws://' + location.hostname,
        'webhuesli-protocol');
```

Internet-Protokoll

returns the hostname of the current URL

subprotocol name,
server accepts only one of passed subprotocols

Weiterführende Infos:

<http://www.html5rocks.com/en/tutorials/websockets/basics/>

WebSocket Events

- WebSockets unterstützen folgende Ereignisse:
 - onopen (öffnen des WebSockets)
 - onmessage (Empfang einer Nachricht)
 - onerror (aufgetretener Fehler)
 - onclose (WebSocket wurde geschlossen)

- Beispiel

```
exampleWebSocket.onopen = function () {  
    alert("Socket has been opened!");  
}
```

WebSocket Zustände

- WebSockets können folgende Zustände annehmen:
 - CONNECTING
 - OPEN
 - CLOSING
 - CLOSED
- Der Zustand kann im Attribut `readyState` abgefragt werden.
- Beispiel

```
if( exampleWebSocket.readyState == OPEN ) {  
    ...  
}
```

Daten senden über WebSockets

- Prototyp

`send(data) ;`

- data: zu sendende Daten (String, JSON-Objekt ...)

- Beispiel

`var data = "Hallo" ;`

`exampleWebSocket.send(data) ;`

Daten empfangen über WebSockets

- Für das Empfangen von Daten wird das Event „onmessage“ verwendet.

- Beispiel

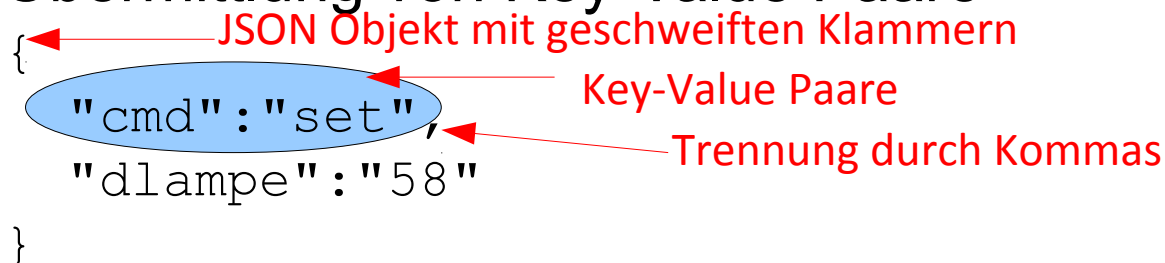
```
exampleWebSocket.onmessage = function(msg) {  
    var data = msg.data;  
    ...  
}
```

WebSockets schliessen

- Prototyp
`close()` ;
 - Keine Argumente
- Beispiel
`exampleWebSocket.close()` ;

JSON (JavaScript Object Notation)

- Kompaktes textbasiertes Datenformat für den Datenaustausch.
- Unabhängig von der Programmiersprache.
- Übermittlung von Key-Value Paare



The diagram shows a JSON object: `{ "cmd": "set", "dlampe": "58" }`. Annotations with red arrows point to specific parts:

- A red arrow points to the opening curly brace `{` with the text "JSON Objekt mit geschweiften Klammern".
- A blue oval highlights the first key-value pair `"cmd": "set"`. A red arrow points to it with the text "Key-Value Paare".
- A red arrow points to the comma `,` after the first pair with the text "Trennung durch Kommas".

- Funktionen von JavaScript, um JSON-Objekte zu verarbeiten. Beispiele:

```
var jsonObject1 = {"cmd": "set", "dlampe": "58"};
var jsonObject2 = JSON.parse("String");
var string = JSON.stringify(jsonObject1);
var dlampevalue = jsonObject1.dlampe;
```