Internetkommunikation mit der Socket-API

- Ein Socket (*Steckdose*) ist ein Kommunikations-Endpunkt.
- Austausch von Nachrichten zwischen Prozessen erfolgt *verbindungsorientiert* (SOCK_STREAM) oder im *Datagramm-Stil* (SOCK_DGRAM)
- Kommunizierende Prozesse können auf demselben Rechner ablaufen oder auf unterschiedlichen miteinander vernetzten Maschinen
- Die Programmierschnittstelle ist generisch, d.h. sie unterstützt verschiedene Protokollfamilien und - daran gekoppelt - verschiedene Adressierungsarten (*Adressdomänen*, *Adressfamilien* oder *Kommunikationsdomänen*).

  Beispiele für Protokollfamilien:

  - *PF_UNIX* – ("UNIX domain sockets"): effiziente rechnerinterne Kommunikation, Adressobjekte sind Pfade im Dateisystem
  - *PF_INET* – Internet V4
  - *PF_INET6* – Internet V6

  Die entsprechenden Adressfamilien sind *AF_UNIX*, *AF_INET* bzw. *AF_INET6*. Die ursprünglich als sinnvoll erachtete Trennung von Adress- und Protokollfamilien wurde bislang für keine Protokollfamilie benötigt, so dass tatsächlich *PF_xxxxx* und *AF_xxxxx* identisch definiert sind.

Internet-Sockets

- verbindungsorientierte Kommunikation mit TCP (Transmission Control Protocol)
- Datagramm-Austausch mit UDP (User Datagram Protocol)
- Adressen bestehen aus der IP-Nummer des Rechners und einer (TCP- oder UDP-) Port-Nummer
- Die IP-Nummer identifiziert einen Netzwerkadapter eines Rechners im Netzwerk
- Die Port-Nummer ist eine innerhalb des Rechners eindeutige Zieladresse für Nachrichten.

Socket-Aufruf

```
#include <sys/types.h>
#include <sys/socket.h>

int socket (int protokollfamilie, int typ, int protokoll);
```

Durch Protokollfamilie und Typ ist meist das Protokoll schon festgelegt, man kann als dritten Parameter in der Regel einfach 0 übergeben und erhält das

Default-Protokoll (für Datagramme: UDP, verbindungsorientiert: TCP).

Aufrufbeispiel:

```
int sock=socket(PF_INET, SOCK_DGRAM, 0);
```

Internet-Adressen

Für die Weiterleitung einer Nachricht im Internet zwischen zwei Rechnern ist die IP-Adresse der jeweiligen Netzwerkschnittstellen maßgeblich, für die Adressierung der richtigen Empfänger-Anwendung innerhalb des Empfänger-Rechners die Portnummer des Transportprotokolls.

Zum Nachrichtenaustausch ist immer eine lokale **und** eine entfernte Adresse nötig. Da der socket-Aufruf keine Adressen festlegt, sind die Adressen durch andere Funktionen festzulegen:

- *bind* legt die lokale Adresse fest

  Dabei kann man wählen, ob die Kommunikation auf eine bestimmte Netzwerkschnittstelle des Rechners beschränkt sein soll oder ob alle Schnittstellen benutzt werden können. Ein Server muss seine lokale Adresse festlegen, damit er erreichbar ist.
- Die Empfängeradresse kann vor dem Versenden mit *connect* oder beim Versenden mit *sendto* oder *sendmsg* festgelegt werden.

Interne Repräsentation von Internetadressen

Eine Adresse der Adressfamilie AF_INET ist vom Typ

```
struct sockaddr_in
```

und besteht aus folgenden Komponenten

Internetadresse

Adressfamilie (`sin_family`)	IP-Nummer (`sin_addr`)	Port (`sin_port`)
AF_INET	**212.201.7.21**	**80**

- `sin_family`: Adressdomäne (= AF_INET)
- `sin_addr`: IP-Nummer des Rechners – 4 Byte

  Der Typ von `sin_addr` ist `struct in_addr`, eine Struktur mit nur einer Komponente `s_addr`.
- `sin_port`: Portnummer – 2 Byte

Aus technischen Gründen können implementierungsabhängig innerhalb der Adressstruktur vor und hinter diesen Komponenten Lücken entstehen. Zur korrekten Verarbeitung sollte man sicherstellen, dass die gesamte Struktur mit Nullen initialisiert wird, z.B.

```
struct sockaddr_in serveradresse;
memset( (void*) &serveradresse, 0, sizeof serveradresse );
```

Portnummern

- Portnummern sind protokollspezifisch, TCP-Port X $\neq$ UDP-Port X
- Eine Portnummer dient rechnerintern als eindeutiges „Nachrichten-Postfach“:
  - in einer entfernten Adresse dient es zur Adressierung der Empfänger-Anwendung
  - in der eigenen Adresse ist ein Nachrichtenpuffer damit assoziiert, in den das Betriebssystem Nachrichten ablegt
- Server für wohlbekannte Dienste verwenden feste, für den Dienst reservierte Portnummern, z.B. 80 für einen WWW-Server (s. /etc/services). Die Nummern 1 bis 1023 sind dafür reserviert.
- Portnummern der Client-Anwendungen sind von den Server-Portnummern unabhängig!
- Ein Programm, das eine beliebige *neue* Portnummer benötigt, übergibt an das Betriebssystem beim bind-Aufruf eine 0 als Port. Das Betriebssystem sucht daraufhin eine noch nicht vergebene Nummer und trägt sie in die Adressstruktur des Socket ein.

IP-Nummern bei mehreren Netzwerkschnittstellen

Spezielle IP-Nummer in der Absenderadresse: INADDR_ANY

- INADDR_ANY in der Absenderadresse beim Empfang: Empfang von allen Schnittstellen
- INADDR_ANY in der Absenderadresse beim Senden: Absender-IP-Nummer wird die Nummer der Schnittstelle, über die (gemäß Routingregeln) versendet wird

Adressdarstellung im Netz und im Rechner: Byte-Anordnung

Problem: Bei der Abspeicherung von mehreren Byte großen binär kodierten Zahlen (z.B. C-Datentypen short, int oder long) ist die Anordnung der einzelnen Bytes im Hauptspeicher vom Rechner abhängig. Bei "little endian"-Rechnern (Intel PC) steht das kleinstwertige Byte an der kleinsten Adresse, bei "big endian"-Rechnern (z.B. ARM, Sun SPARC) ist es umgekehrt.

Wenn man eine solche Zahl im binären Format in einer Netzwerk-Nachricht verpackt versendet, müssen Sender und Empfänger sich dabei ggf. auf eine gemeinsame Byte-Anordnung einigen. In Internet-Paketen werden auch die Absender- und Empfängeradressen binär kodiert. Damit ein Router diese Adressen korrekt verarbeitet kann, muss er ebenfalls die Byte-Anordnung kennen.

Beispiel: Eine Variable i vom Typ int ist 4 Byte groß, hat die (Anfangs-)Adresse 1000 und den Wert 1.

Byte-Adresse	**Byte-Inhalt**	
	"little endian"	"big endian"
1000	0x01	0x00
1001	0x00	0x00
1002	0x00	0x00
1003	0x00	0x01

Einheitliches Netzwerkformat für IP-Adressen

Für TCP- und UDP Portnummern, sowie für IP-Adressen wird im Internet die "big endian"-Darstellung verwendet.

Konversionsroutinen für Portnummern (`unsigned short integer`) und IP-Nummern (`unsigned long integer`) vom Rechnerformat in das Netzformat und zurück:

`htons`	host-to-net short
`htonl`	host-to-net long
`ntohs`	net-to-host short
`ntohl`	net-to-host long

Beim Binden von Adressen an Sockets ist in jedem Fall das Netzwerkformat zu verwenden. Wenn also die Portnummer 80 eines WWW-Servers adressiert werden soll, darf in die Adressstruktur nicht einfach `80` eingesetzt werden, sondern `htons(80)`.

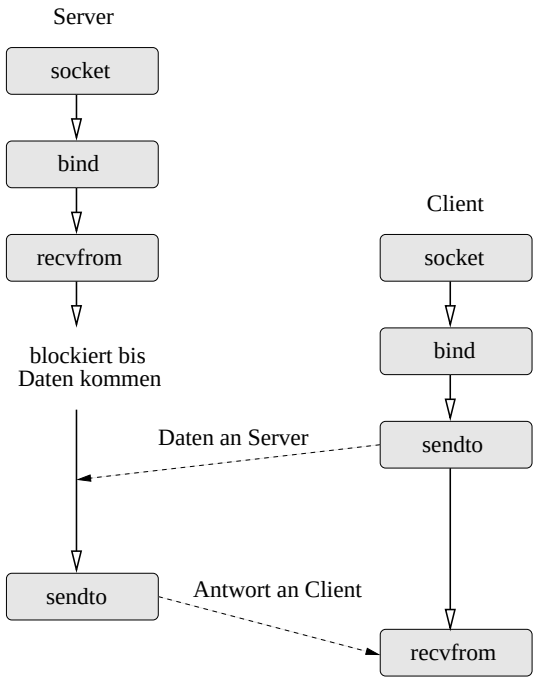
Externe Repräsentation der IP-Nummern

Ein anderes Problem ist die externe und interne Repräsentation der IP-Nummern. Um die Nummer besser handhaben zu können, werden diese 4-Byte-Adressen Byte-weise dezimal mit Punkten als Trennzeichen geschrieben, z.B. "212.201.7.21".

Mittels Konversionsroutinen lassen sich solchen zu solchen XSZeichenketten die entsprechenden internen Repräsentationen erstellen und umgekehrt. Konversionsroutinen, die sowohl für IPv4 als auch IPv6 funktionieren sind *inet_pton* und *inet_ntop*, ältere Funktionen, die nur IPv4 unterstützen, sind *inet_aton* und *inet_ntoa*. Details stehen im System-Manual.

Datagramm-Kommunikation: bind, sendto und recvfrom

Client-Server-Kommunikationsschema



`bind` – Zuordnung einer Adresse zu einem Socket

Mit einem *bind*-Aufruf wird einem Socket eine Internetadresse zugeordnet, nämlich die eigene Adresse, die als Absenderadresse in den versendeten Nachrichten enthalten ist.

Bei Datagramm-Kommunikation müssen beide Kommunikationspartner eine Adresse an ihre Sockets binden.

```
#include <sys/types.h>
```

```
#include <sys/socket.h>

int bind(int sockfd,  struct sockaddr *my_addr, int addrlen);
```

Der zweite Parameter ist die Adresse, die an den Socket gebunden wird. Da der *bind*-Aufruf für alle Adressfamilien verwendet wird, ist der Adresstyp `struct sockaddr` generisch definiert. Für jede Adressfamilie gibt es eine spezifische Variante, bei AF_INET: `struct sockaddr_in`.

`addrlen` gibt die Größe der Adresse an, die ja je nach Adressfamilie variieren kann (bei AF_UNIX ist die Adresse ein beliebig langer Dateipfad).

Beispiel:

```
struct sockaddr_in serveradresse, clientadresse;
int sockfd;
...

if ((sockfd=socket(AF_INET, SOCK_DGRAM, 0)) < 0){
  perror("Fehler beim socket-Aufruf");
  exit(1);
  }

/* Adresse mit Nullen initialisieren */
memset( (void*) &serveradresse, 0, sizeof serveradresse);

serveradresse.sin_family=AF_INET;
serveradresse.sin_port=htons(80);
serveradresse.sin_addr.s_addr=inet_aton("212.201.7.21");



clientadresse.sin_family=AF_INET;
clientadresse.sin_port=htons(0);
clientadresse.sin_addr.s_addr=htonl(INADDR_ANY);

if (   bind (sockfd,
             (struct sockaddr *) &clientadresse,
             sizeof(clientadresse)
            ) < 0  ) {
  perror("Fehler beim bind-Aufruf");
  exit(1);
  }
```

...

Nachrichten über Socket verschicken

- Für das Versenden von Nachrichten stehen mehrere Funktionen zur Auswahl
  - write - einfachste Operation (Empfängeradresse vorher mit *connect* festlegen)
  - send - gegenüber *write* einige Zusatzoptionen (z.B. Out-of-Band-Nachrichten)
  - sendto - mit Empfängeradressangabe
  - sendmsg - mit voller Kontrolle über das Nachrichtenformat

sendto

```
#include <sys/types.h>
#include <sys/socket.h>

int sendto(int sockfd,
           const void *nachricht, int laenge_nachricht,
           unsigned int flags,
           const struct sockaddr *empfaengeradresse,
           int laenge_empfaengeradresse);
```

- Sender muss zunächst einen Socket `sockfd` erzeugt und mit `bind` an seine eigene Adresse gebunden haben
- Dann ist die Empfängeradress-Struktur `empfaengeradresse` auszufüllen
- Die im dritten Parameter übergebbaren Flags dienen Sonderfunktionen, wie etwa das Senden sogenannter „Out-of-Band"-Daten (vgl. `sendto`-Manual).
- Der Resultatswert gibt die Anzahl der gesendeten Byte an.

`recvfrom` – Nachricht über Socket lesen

Wie beim Versenden stehen auch beim Empfangen von Nachrichten mehrere Funktionen zur Auswahl: *read* und *recv* liefern nur die Daten, aber keine Absenderadresse, während *recvfrom* und *rcvmsg* zusätzlich die Absenderadresse liefern. Wir beschreiben nur *recvfrom* genauer.

```
#include <sys/types.h>
#include <sys/socket.h>
```

```
int recvfrom(int sockfd,
             void *puffer, int laenge_puffer,
             unsigned int flags
             struct sockaddr *senderadresse,
             int *laenge);
```

- Sobald ein Socket `sockfd` von einem Prozess erzeugt und mit `bind` an seine eigene Adresse gebunden wurde, kann dieser Prozess über den Socket Nachrichten empfangen. `recvfrom` blockiert solange, bis eine Nachricht für den Empfänger-Port eintrifft.
- Die Nachricht wird in einem Puffer hinterlegt (`puffer`), genauso wie die Absenderadresse (`senderadresse`) und deren Länge. Mit diesen Angaben ist der Empfänger in der Lage, eine Antwort zu verschicken. Der Resultatswert der Funktion gibt an, wieviele Byte empfangen wurden.
- **Achtung**: Der letzte Parameter *laenge* ist eine Ein-/Ausgabeparameter. Bei Aufruf ist hier die Größe des für die Absenderadresse reservierten Speicherbereichs zu übergeben. Wenn die tatsächliche Absenderadresse grösser sein sollte als der dafür reservierte Platz, wird nämlich vom Betriebssystem der nicht mehr abspeicherbare Rest abgeschnitten. Dies macht z.B. bei den Adressen der UNIX-Domain Sinn (Dateipfade unterschiedlicher Länge).
- Die Flags dienen Sonderfunktionen (z.B. Lesen von „Out-of-Band“-Daten (vgl. `recvfrom`-Manual).

Client-Server-Schema

Das Kommunikationsschema bei verbindungsorientierter Kommunikation, die innerhalb der AF_INET-Domain auf dem TCP-Protokoll basiert, ist asymetrisch: Ein Kommunikationspartner spielt den Server, andere sind dessen Clients.

Die Nachrichten, die ein Client an den Server schickt, sind beliebig, wir bezeichnen Sie aber im folgenden als *Aufträge*.

Verbindungsaufbau

Der Server benutzt zur Verbindungsaufname die folgenden Systemaufrufe

`socket`	Socket erzeugen
`bind`	Serveradresse an Socket binden
`listen`	Auftrags-Warteschlage initialisieren
`accept`	Auf Auftrag warten

Der Client baut seinerseits die Verbindung auf mit

`socket`	Socket erzeugen
`connect`	Verbindung zum Server aufbauen

Das Concurrent Server- Prinzip

Typischerweise wird der Server als *concurrent server* agieren:

- Ein Thread des Servers, nennen wir ihn „Master-Thread“, wartet auf Aufträge. Diese werden über den „Auftragseingangs-Socket“, der mit den *socket*- und *bind*-Aufrufen initialisiert wurde, übermittelt.
- Bei Eingang eines Auftrags erzeugt der Master-Thread einen neuen „Slave-Thread“, der den Auftrag bearbeitet (klassischerweise ein Subprozess). Währenddessen kümmert sich der Master-Thread schon um den nächsten Auftrag.
- „Auftragseingangs-Socket“ dient der Auftragsannahme.
  Wie kommuniziert der Slave-Thread mit dem Client?
- *accept*-Semantik:

  accept-Aufruf erzeugt für die Bearbeitung des eingegangenen Auftrags automatisch einen weiteren Socket und eine neue Portnummer. Er bindet den neuen Socket an die neue Portnummer.
- Slave-Thread kann über neuen Socket zur Auftragsabwicklung mit Client kommunizieren

Client-Seite

connect-Aufruf

- kehrt zurück, sobald beim Server der neue Kommunikationsport eingerichtet wurde
- bindet Client-Socket an Kommunikationsport des Servers
- erzeugt gleichzeitig auch beim Client einen Port und bindet diesen an dessen Socket (*bind*-Aufruf beim Client überflüssig)

Datenaustausch

Für den Datenaustausch nach erfolgtem Verbindungsaufbau stehen diverse Funktionen zur Verfügung. Im einfachsten Fall verwenden Client und Server *read* und *write* in Verbindung mit den Socket-Deskriptoren.

Der Verbindungsabbau erfolgt mit *close*.

Für Sonderfunktionen lassen sich aber auch folgende Funktionen verwenden: *send*, *sendto*, *sendmsg*, *recv*, *recvfrom*, *recvmsg*.

Welche zusätzliche Funktionalität haben diese Nachrichtenübertragungsfunktionen gegenüber *read* bzw. *write*? Sie bieten z.B. die Möglichkeit, Out-of-Band-Daten zu übermitteln oder Nachrichten zu empfangen, ohne diese aus dem Nachrichten-Eingangspuffer zu entfernen. Daneben gibt es sehr spezielle Funktionen, wie etwa die Übertragung von Dateideskriptoren.