SOCKET-PROGRAMMIERUNG

- Application Programmer Interface (API)
- Design und Strukturierung von Server- und Client-Programmen

Sockets

- Sockets sind eine Abstraktion für Interprozeß-Kommunikation auf Betriebssystem-Ebene
  - Sowohl zur Kommunikation mit lokalen Prozessen (auf dem selben Rechner) als auch mit Prozessen auf entfernten Rechnern
  - Ursprünglich mit 4.2 BSD UNIX eingeführt
  - Heute auf allen wesentlichen Betriebssystemen verbreitet
    - ◇ auch auf Windows bzw. Win32 Systemen
- Sockets wurden ursprünglich für die TCP/IP Protokolle entwickelt.
  - Später wurde das Socket-Konzept generalisiert, um auch andere Protokoll-Familien zu unterstützen.
  - Beispiele: Novell IPX/SPX, ATM
- Socket-Routinen kontrollieren die Kommunikation zwischen Prozessen und Protokollen.
  - Außerdem stellen sie Puffer bereit zwischen der *synchronen* Applikations-Domäne und der *asynchronen* Betriebssystem-Domäne.

Das Material dieses Kapitels basiert auf Kursunterlagen von Prof. Douglas C. Schmidt, University of California at Irvine, USA.

(`http://www.ece.uci.edu/~schmidt/`)

Communication Domains

- Communication domains are a key structuring concept in the BSD networking architecture
  - e.g. Internet domain and UNIX domain
- Domains specify:
  - The scope over which two processes may communicate
    - ◇ e.g. local only vs. local/remote
  - How names and addresses are formed and interpreted in subsequent socket calls
    - ◇ e.g. pathnames vs. IP/port numbers
- Most socket implementations provide several domains represented as 'protocol families'
  - The socket interface is used for all these protocol family domains
- UNIX domain (PF_UNIX)
  - Communicate only with a process on the same machine
    - ◇ Uses UNIX filenames for rendezvous between client and server processes
  - Really a form of intra-machine IPC, similar to SVR4 STREAM pipes
    - ◇ Supports both reliable (SOCK_STREAM) and unreliable (SOCK_DGRAM) local IPC
    - ◇ Used for local X-window traffic...
  - UNIX domain sockets have been used originally for implementing pipes.
    - ◇ Although still available, UNIX domain sockets have been replaced by STREAM pipes.
- Internet domain or TCP/IP (PF_INET)
  - Communicate across network or on same machine (uses "dotted-decimal Internet addresses")
    - ◇ e.g. "128.195.1.1 at port 21"
  - e.g. TCP, UDP, IP, ftp, rlogin, telnet
- Xerox XNS (later evolved into Novell IPX) (e.g. SPP, PEX, IDP)
- ISO/OSI (e.g. TP4-TP1, CLNS, CONS)

Socket Types

- There are five types of sockets
  - Stream Socket
    - ◇ Reliable (sequenced, non-duplicated, non-corrupted) bi-directional delivery of byte-stream data
    - ◇ Metaphor: A "network pipe"
    - ◇ Example
      ```
      int s = socket (PF_INET, SOCK_STREAM, 0);
      /* Note, s is an internal id... */
      ```
  - Datagram Socket
    - ◇ Unreliable, unsequenced datagram
    - ◇ Metaphor: Sending a letter
    - ◇ Example
      ```
      int s = socket (PF_INET, SOCK_DGRAM, 0);
      ```
  - Reliably-delivered Message Socket
    - ◇ Reliable datagram
    - ◇ Metaphor: Sending a registered letter
    - ◇ Example
      ```
      int s = socket (PF_NS, SOCK_RDM, 0);
      ```
  - Sequenced Packet Stream Socket
    - ◇ Reliable, bi-directional delivery of record-oriented data
    - ◇ Metaphor: Record-oriented TCP (e.g. TP4 and XTP)
    - ◇ Example
      ```
      int s = socket (PF_NS, SOCK_SEQPACKET, 0);
      ```
  - Raw Sockets
    - ◇ Allows user-defined protocols that interface with IP
    - ◇ Requires root access
    - ◇ Example
      ```
      int s = socket (PF_INET, SOCK_RAW, 0);
      ```
- SOCK_STREAM and SOCK_DGRAM are the most common types of sockets...

Socket Addresses

- UNIX supports multiple communication domains, protocol families, and address families
  - The socket API provides a single address interface for all these families
- The type of the sockaddr structure used with accept, bind, connect, sendto, and recvfrom differs according to the domain (UNIX vs. Internet vs. XNS)
- The addressing API has a somewhat confusing and error-prone design
  - Motivation was to save space for the "common case"...
- General Format
  ```
  struct sockaddr { u_short sa_family; char sa_data[14]; };
  ```
- UNIX Domain
  ```
  struct sockaddr_un {
      short sun_family; char sun_path[108];
  };
  ```
- Internet Domain
  ```
  struct in_addr { unsigned long s_addr; };
  struct sockaddr_in {
      short sin_family; u_short sin_port;
      struct in_addr sin_addr; char sin_zero[8];
  };
  ```
- General usage for Internet-domain service:
  ```
  struct sockaddr_in addr;
  memset (&addr, 0, sizeof addr);
  addr.sin_family = AF_INET;
  addr.sin_port = htons (port_number);
  addr.sin_addr.s_addr = htonl (INADDR_ANY);
  if (bind (sock_desc, (struct sockaddr *) &addr, sizeof addr)
        == -1)
     ...;
  ```
- Note the use of a cast
  - In C++, this whole mess can be cleaned-up via inheritance and dynamic binding!

Socket Operations

- Local context management
  - int socket (int domain, int type, int protocol);
  - int bind (int fd, struct sockaddr *, int len);
  - int listen (int fd, int backlog);
  - int close (int fd);
  - int getpeername (int fd, struct sockaddr *, int *len);
  - int getsockname (int fd, struct sockaddr *, int *len);
- Connection establishment and termination
  - int connect (int fd, struct sockaddr *, int len);
  - int accept (int fd, struct sockaddr *, int *len);
  - int shutdown (int fd, int how);
- Option management
  - int ioctl (int fd, int request, char *arg);
  - int fcntl (int fd, int cmd, int arg);
  - int getsockopt (int, int, int, char *, int *);
  - int setsockopt (int, int, int, char *, int);
- Data transfer
  - int read (int fd, void *buf, int len);
  - int write (int fd, void *buf, int len);
  - int send (int fd, void *buf, int len, int flags);
  - int recv (int fd, void *buf, int len, int *flags);
  - int readv (int fd, struct iovec [], int len);
  - int writev (int fd, struct iovec [], int len);
  - int sendto (int fd, void *buf, int len, int flags, struct sockaddr *, int len);
  - int recvfrom (int fd, void *buf, int len, int flags, struct sockaddr *, int *len);
  - int sendmsg (int fd, struct msghdr *msg, int flags);
  - int recvmsg (int fd, struct msghdr *msg, int flags);

- Event demultiplexing
  - int select (int maxfdp1, fd_set *rdfds, fd_set *wrfds, fd_set *exfds, struct timeval *);
- Byte order conversion
  - u_short htons (u_short hostshort);
  - u_short ntohs (u_short netshort);
  - u_long htonl (u_long hostlong);
  - u_long ntohl (u_long netlong);

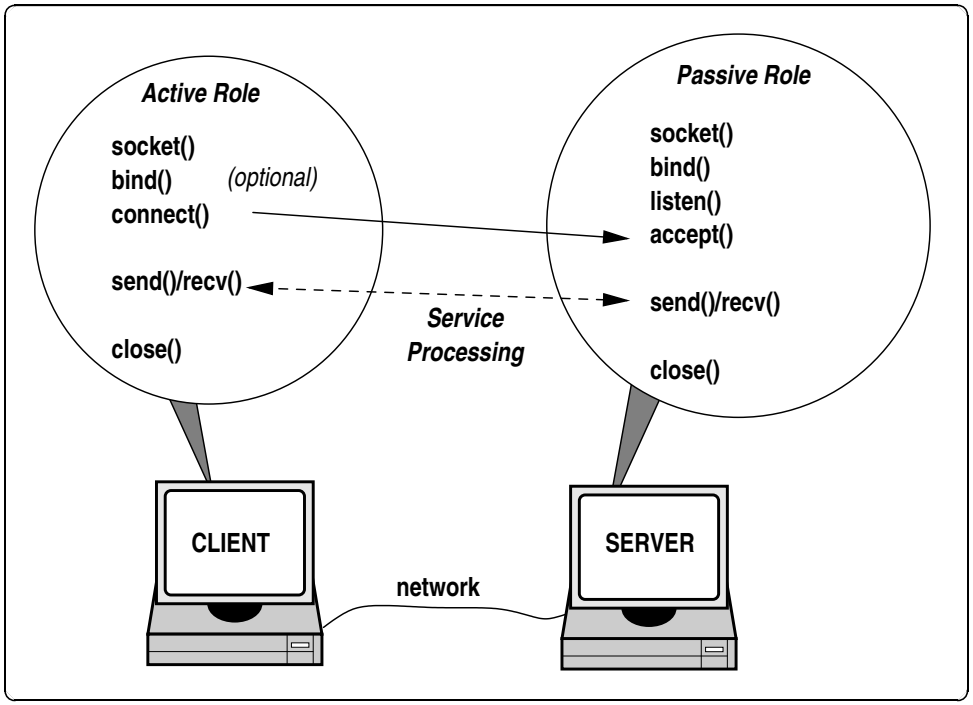

Abb. PRG–1 Verbindungs-orientierter Einsatz von Sockets

- Abb. PRG–1 zeigt typische Aufruf-Sequenzen von Socket-Routinen für Verbindungs-orientierten Einsatz von Sockets.

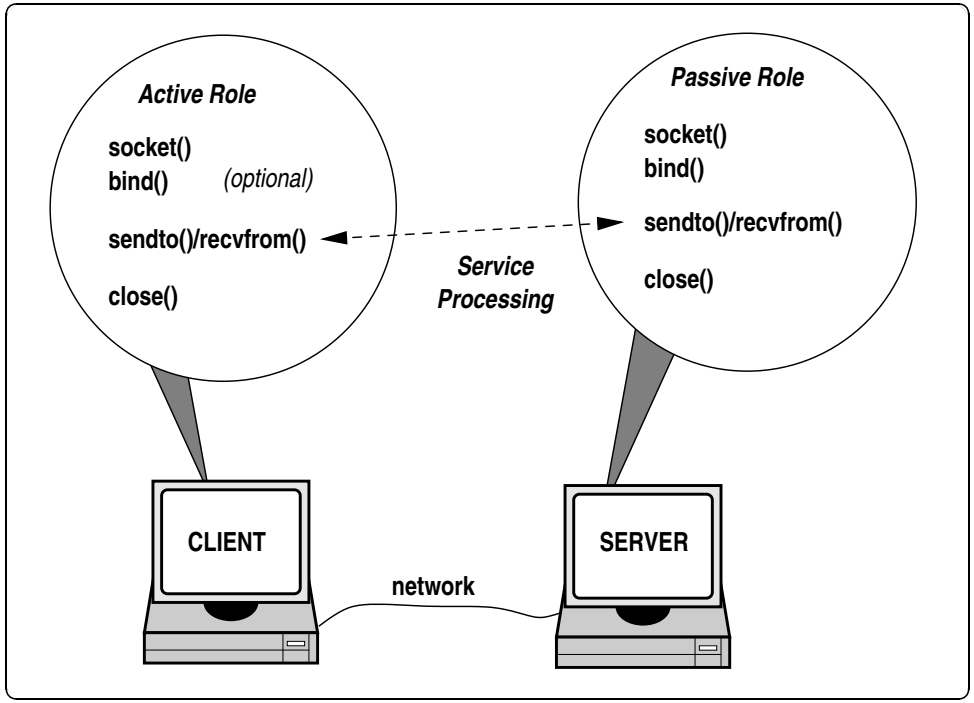

Abb. PRG–2 Verbindungsloser Einsatz von Sockets

- Abb. PRG–2 zeigt typische Aufruf-Sequenzen von Socket-Routinen für verbindungslosen Einsatz von Sockets.

Operations of Client and Server

- `socket`
  - Creates and opens a socket and returns a descriptor
  - int s = socket (int domain, int type, int protocol);
    - *domain* → PF_UNIX, PF_INET
    - *type of service* → SOCK_STREAM, SOCK_DGRAM
    - *protocol* → generally 0, but could be TCP, VMTP, NETBLT, XTP

- `bind`
  - Associates a local ("the own") address (e.g. an IP address, address family, and port number) to an unnamed socket
  - int bind (int s, struct sockaddr *addr, int addrlen);
  - *addr* → local address
    - e.g. points to an Internet addr or a UNIX domain addr
  - *addrlen* → length of address
  - Note
    - `bind` is not necessary for clients (which implicitly allocate transient port numbers)
    - The address INADDR_ANY is a wildcard for any server host/network interface
    - Always "zero-out" the address structure before using it...

- `close`
  - Close a socket
  - int close (int s);
    - Note, there are subtle semantics related to "grace termination..." of protocols

- `shutdown`
  - Shutdown part or all of full-duplex connection
  - int shutdown (int s, int how);
    - *how* is 0, then further receives will be disallowed
    - *how* is 1, then further sends will be disallowed
    - *how* is 2, then further sends and receives will be disallowed
  - Note, `shutdown` does *not* close the descriptor...

- `getsockname`
  - Returns address info describing the local socket `s`
  - int getsockname (int s, struct sockaddr *addr, int *addrlenptr);
    - *addr* → address of local binding
    - *addrlenptr* → ptr to length of address

- `getpeername`
  - Returns the current *name* for the specified connected peer socket
  - int getpeername (int s, struct sockaddr *addr, int *addrlenptr);
    - *addr* → address of remote peer
    - *addrlenptr* → ptr to length of address

Typical Client Operations

- `connect`
  - Specify foreign/remote destination address (e.g. IP/port numbers) and joins two sockets for I/O
  - int connect (int s, struct sockaddr *addr, int addrlen);
    - *addr* → address of remote client
    - *addrlen* → length of address

Typical Server Operations

- `listen`
  - Set the length of a TCP passive open queue, places the socket into "passive-mode"
  - This tells kernel to accept connection requests for a listening socket on behalf of a client
  - int listen (int s, int backlog);
    - *backlog* → specifies how many connection requests can be queued
  - Note, the kernel will queue a certain number of incoming connection requests on behalf of the server
    - Otherwise, pending requests would be dropped due to finite limits on OS queue sizes...
    - These limits prevent "denial of service" attacks...

- `accept`
  - Returns a unique descriptor to the next available completed connection from the connection queue
  - int accept (int s, struct sockaddr *addr, int *addrlenptr);
    - *addr* → address of remote client
    - *addrlenptr* → ptr to length of address
    - Returns new socket descriptor specifying the full association
  - Notes:
    - Server may decide to reject connection only after first accepting it!
    - *addr* and *addrlenptr* may be 0...

- `select`
  - Synchronous event demultiplexer that queries the status of a set of socket descriptors under timer control
  - int select (int maxfdp1, fd_set *readfds, fd_set *writefds,
    fd_set *exceptfds, struct timeval *timeout);
    - *maxfdp1* → max file descriptor to consider plus 1
    - *readfds* → set of descriptors to check for reading and incoming connections
    - *writefds* → set of descriptors to check for writing and outgoing connections
    - *exceptfds* → set of descriptors to check for urgent data
    - *timeout* → length of time to wait for activity on the descriptors

Data Transfer Operations

- `write`
  - Send a message to a socket
  - int write (int s, char *msg, int len);
    - *msg* → buffer of data to send
    - *len* → length of buffer

- `send`
  - Send a message to a socket
  - int send (int s, char *msg, int len, int flags);
    - *flags*
      - MSG_OOB → send *out-of-band* data on sockets that support this operation

- Note that neither `write` nor `send` are guaranteed to write all the bytes!

- `read`
  - Receive a message from a socket
  - int read (int s, char *buf, int len);

- `recv`
  - Receive a message from a socket
  - int recv (int s, char *buf, int len, int flags);
    - *flags*
      - MSG_OOB → read any *out-of-band* data present on the socket, rather than the regular *in-band* data
      - MSG_PEEK → *Peek* at the data present on the socket; the data are returned, but not consumed, so that a subsequent receive operation will see the same data

- `sendto`
  - Send a datagram message to a UDP socket
  - int sendto (int s, char *msg, int len, int flags,
    struct sockaddr *addr, int addrlen);
    - *addr* → address of remote server
    - *addrlen* → length of address

- `recvfrom`
  - Receive a datagram message from a UDP socket
  - int recvfrom (int s, char *buf, int len, int flags,
    struct sockaddr *addr, int *addrlenptr);
    - *addr* → address of remote server
    - *addrlenptr* → ptr to length of address
      **addr* and **addrlenptr* are set to values corresponding to the actual sender of the datagram

Byte order conversion

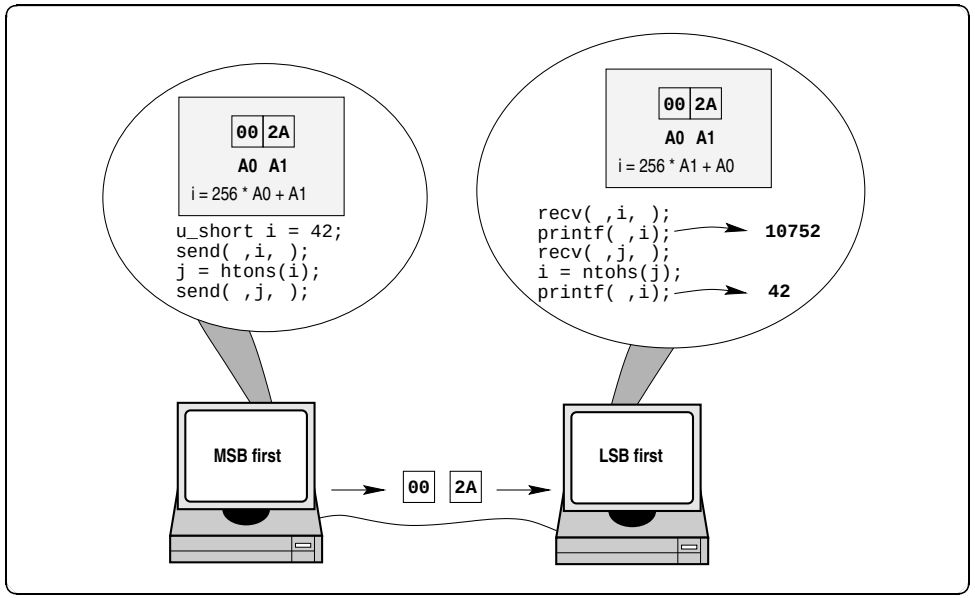

Abb. PRG–3 The problem with different byte orders.

- `htons`
  - Convert an unsigned short (16-bit) integer from host byte order to Internet network-byte order.
- `ntohs`
  - Convert an unsigned short (16-bit) integer from Internet network-byte order to host byte order.
- `htonl`
  - Convert an unsigned long (32-bit) integer from host byte order to Internet network-byte order.
- `ntohl`
  - Convert an unsigned long (32-bit) integer from Internet network-byte order to host byte order.

Option Management

- `setsockopt`
  - Sets options on a socket
  - int setsockopt (int s, int level, int optname, void *optval, int optlen);
- `getsockopt`
  - Gets options regarding a socket
  - int getsockopt (int s, int level, int optname, void *optval, int *optlenptr);
- Arguments for `setsockopt` and `getsockopt`
  - *level* → protocol level (e.g. IP, TCP, socket, etc.)
    - ◇ e.g. SOL_SOCKET, IPPROTO_TCP, IPPROTO_IP
  - *optname* → name of option
    - ◇ e.g. SO_REUSEADDR, SO_ERROR, SO_BROADCAST, SO_SNDBUF, SO_RCVBUF
  - *optval* → value of option
  - *optlen* → length of option

Auxiliary Networking Functions

- `gethostname`
  - Returns the primary name of the current host as an ASCII string
  - int gethostname (char *name, int namelen);
- `gethostbyname/gethostbyaddr`
  - struct hostent *gethostbyname (char *name);
  - struct hostent *gethostbyaddr (char *, int len, int type);
- These routines interface with local configurations and with the DNS domain name service.

Example: Internet Domain Stream Sockets

- Auxiliary routine, copying data from ifd to ofd

```
int process_msg (int ifd, int ofd) {
   for (char msg[BUFSIZ];;) {
      ssize_t len = read (ifd, msg, sizeof msg);
      if (len > 0) {
         if (send_n (ofd, msg, len) != len)
            return -1;
      } else return len;
   }
   return 0;
}
```

- send_n is a handy utility routine

```
ssize_t send_n (int handle, const void *buf, size_t len) {
   size_t bytes_written;
   ssize_t n;

   for (bytes_written = 0; bytes_written < len;
         bytes_written += n)
      if ((n = write (handle, buf + bytes_written,
                      len - bytes_written)) == -1)
         return -1;
   return bytes_written;
}
```


- File header.h:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <string.h>
#include <netinet/in.h>
#include <netdb.h>

#define SRV_PORT 7734
#define SRV_ADDR "128.195.13.4"
#define STDOUT 1
#define STDIN 0
```

- Server main program

```
#include "header.h"
int main (int argc, char *argv[]) {
   int s_fd = s_server (SRV_PORT);

   if (s_fd == -1)  perror ("s_server");
   for (;;) {
      int cli_fd = accept (s_fd, 0, 0);

      if (cli_fd == -1)
         perror ("accept");
      else if (process_msg (cli_fd, STDOUT) == -1)
         perror ("process_msg");
      else if (close (cli_fd) == -1)
         perror ("close");
   }
   /* NOTREACHED */
}
```

- Become a passive-mode "server"

```
int s_server (unsigned short port) {
   struct sockaddr_in name;
   memset ((void *), &name, 0, sizeof name);
   name.sin_family = AF_INET;
   name.sin_port = htons (port);
   name.sin_addr.s_addr  = htonl (INADDR_ANY);

   int s_fd = socket (PF_INET, SOCK_STREAM, 0);
   if (s_fd == -1)
      return -1;
   else if (bind (s_fd, &name, sizeof name) == -1)
      return -1;
   else if (listen (s_fd, 5) == -1)
      return -1;
   return s_fd;
}
```

- Client main program

```
#include "header.h"
int main (int argc, char *argv[]) {
   int status = 1;
   int s_fd = s_client (SRV_PORT, SRV_ADDR);
   if (s_fd == -1)
      perror ("s_client");
   else if (process_msg (STDIN, s_fd) == -1)
      perror ("process_msg");
   else
      status = 0;
   close (s_fd);
   return status;
}
```


- Become an active-mode "client"

```
int s_client (u_short port, const char *addr) {
   struct sockaddr_in name;

   memset ((void *) &name, 0, sizeof name);
   name.sin_family = AF_INET;
   name.sin_port = htons (port);
   name.sin_addr.s_addr = inet_addr (addr);
         // inet_addr() translates an Internet network
         // address string to an Internet address integer

   int s_fd = socket (PF_INET, SOCK_STREAM, 0);

   if (s_fd == -1)
      return -1;
   else if (connect (s_fd, (struct sockaddr *) &name,
                     sizeof name) == -1)
        return -1;
   return s_fd;
}
```

Example: Concurrent Server using select

- Single-threaded concurrent socket server
  - Deals with sets of file descriptors.
  - Macros `FD_ZERO, FD_SET, FD_CLR` , and `FD_ISSET` manipulate these sets.
  - File descriptors (for sockets) are assigned numbers in increasing order.
- Example server implements a centralized server for logging records.

```
int main (void)
{
  // Create a server end-point.
  int s_fd = s_server (PORT_NUM);
  fd_set temp_fds;
  fd_set read_fds;
  int maxfdp1 = s_fd + 1;  // initialize

  // Check for constructor failure.
  if (s_fd == -1)
    perror ("server"), exit (1);

  FD_ZERO (&temp_fds);
  FD_ZERO (&read_fds);
  FD_SET (s_fd, &read_fds);
```



```
// Loop forever performing logging server processing.
for (;;) {
  temp_fds = read_fds; // Structure assignment.

  // Wait for client I/O events (handle interrupts).
  while (select (maxfdp1, &temp_fds, 0, 0, 0) == -1
         && errno == EINTR)
    continue;

  // Check for incoming connections.
  if (FD_ISSET (s_fd, &temp_fds)) {
    static struct timeval poll_tv = {0, 0};

    // Handle all pending connection requests
    // (note use of "polling" feature).
    while (select (s_fd + 1, &temp_fds,
                   0, 0, &poll_tv) > 0) {
      int cli_fd = accept (s_fd, 0, 0);

      if (cli_fd == -1) perror ("accept");
      else {
        FD_SET (cli_fd, &read_fds);
        if (cli_fd >= maxfdp1)
          maxfdp1 = cli_fd + 1;
      }
    }
  }
```

```
    // Handle pending logging records (s_fd + 1
    // is guaranteed to be lowest client descriptor).
    for (int fd = s_fd + 1; fd < maxfdp1; fd++)
      if (FD_ISSET (fd, &temp_fds)) {
        int n = handle_logging_record (fd);
        // Must guarantee not to block in this case!
        if (n == -1)
          perror ("logging failed");
        else if (n == 0) {
          // Handle client connection shutdown.
          FD_CLR (fd, &read_fds);
          close (fd);
          if (fd + 1 == maxfdp1) {
            // Skip past unused descriptors.
            while (!FD_ISSET (--fd, &read_fds))
              continue;
            maxfdp1 = fd + 1;
          }
        }
      }
    }
  }
}
```


Example: Internet Domain Datagram Sockets

- Uses UDP to return the current time of day from a specified list of Internet hosts
- E.g.
  - % hostdate tango mambo lambada merengue
  - tango: timeout at host
  - mambo: Tue Aug 20 15:55:59 1996
  - lambada: Tue Aug 20 15:55:59 1996
  - merengue: Tue Aug 20 15:56:00 1996
- Note the use of `select` to prevent hanging from hosts that are "down" or non-existent
- Main driver program

```
#define SERVICE "daytime"
int do_service (int, u_short, const char *);
int main (int argc, char *argv[]) {
   int s = socket (PF_INET, SOCK_DGRAM, 0);
   if (s == -1)   perror (argv[0]), exit (1);

   struct servent *sp = getservbyname (SERVICE, "udp");
   if (sp == 0) {
      fprintf (stderr, "%s/udp: unknown service.\n",SERVICE);
      exit (1);
   }
   for (++argv ;--argc; ++argv)
      if (do_service (s, sp->s_port, *argv) == -1)
         perror (*argv);
   close (s);
   return 0;
}
```

```
int do_service (int sfd, u_short port, const char *host) {
   struct hostent *hp = gethostbyname (host);
   if (hp == 0) return -1;
   struct sockaddr_in sin;
   sin.sin_family = AF_INET;
   sin.sin_port   = port;
   memset (&sin.sin_addr, hp->h_addr, hp->h_length);
   printf ("%s: ", host); fflush (stdout);
   char buf[BUFSIZ];
   if (sendto (sfd, "", 0, /* Note zero size! */
               0, &sin, sizeof sin) < 0)
      return -1;
   struct timeval tv = {5, 0};
   int len = sizeof sin;
   ssize_t n = timed_recv (&tv, sfd, buf, sizeof buf, &sin, &len);
   if (n == -1) return n;
   printf ("%*s\n", n, buf);
   return 0;
}

int timed_recv (struct timeval *tv, int fd,
                char buf[], int buf_size,
                struct sockaddr *sin, int *slen) {
   fd_set read_fd;   FD_ZERO (&read_fd);   FD_SET (fd, &read_fd);
   switch (select (fd + 1, &read_fd, 0, 0, tv)) {
      case 0 : errno = ETIMEDOUT; /* FALLTHRU */
      case -1: return -1;
      default:
         return recvfrom (fd, buf, buf_size, 0, &sin, &slen);
   }
}
```


Advanced Socket Operations

- Non-blocking connections
  - `connect` may be used in non-blocking mode
  - A combination of `select`, `getpeername`, and `getsockopt` may be used to determine when the connection setup is complete
  - This is useful to avoid long timeouts if server may not be accessible

- Checking for invalid sockets
  - It is often useful to have the client test if a previously established socket is still active before trying to write to it
    - ◇ This avoids catching SIGPIPE and such...
  - To do this, first try to `read` from the socket
    - ◇ If the client has closed the connection the `read` should return EOF
  - To keep from hanging in `read`, first put the socket descriptor in non-blocking mode
    - ◇ Conversely, use `select` to find out whether `read` will block...

- Checking for terminated peers
  - A question that often arises is "how do I get the first write to generate SIGPIPE after the other end has terminated?"
  - The answer is "you can not"
  - If you want to know as soon as the process at the other end of a connection terminates, use `select()`, testing for readability, then the `read` will return 0

Design von Netzwerk-Software

Entwurf von Server-Programmen

- Designziele:
- Server-Programm ("Daemon") soll in der Lage sein, mehrere Clients gleichzeitig zu bedienen.
- Clients (und deren Anforderungen an den Server) sollen sich gegenseitig nicht stören.
  - Möglichst gleichzeitige/parallele Bearbeitung
- Der Server soll ökonomisch mit den System-Ressourcen seiner Maschine umgehen.
  - Rechenleistung, Hauptspeicher, Prozeß-Tabelle ...

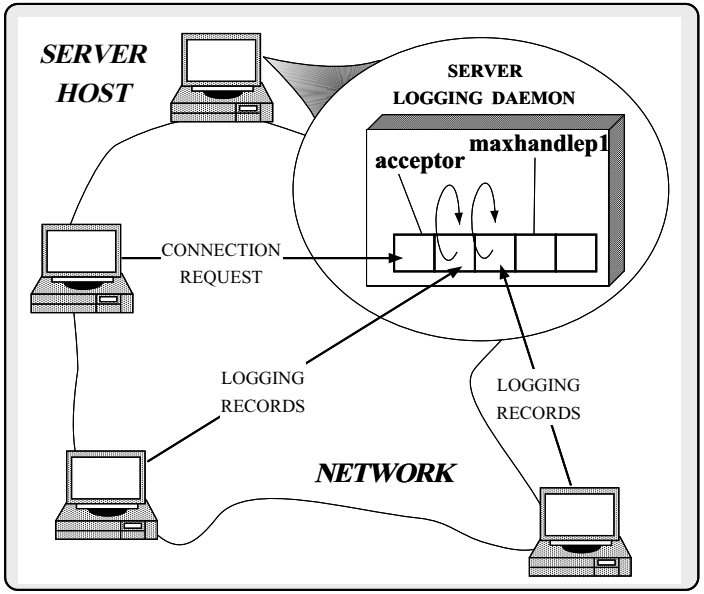

Abb. PRG–4 Server, basierend auf Polling mit nicht-blockierender I/O

- Polling mit nicht-blockierender I/O
  - Vorteile:
    - ◇ Gute Portabilität über diverse UNIX- bzw. PC-Plattformen
  - Nachteile:
    - ◇ Ineffizient (Vergeudung von CPU-Zeit)
    - ◇ Nicht erweiterbar auf weitere Arten von Ereignissen oder Services ohne Umschreiben des Server-Codes

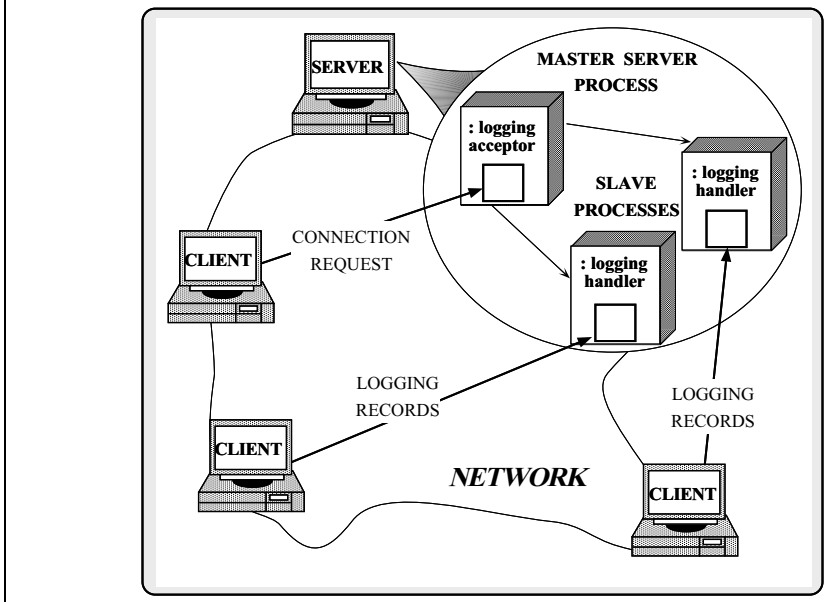

Abb. PRG–5 Server, basierend auf Prozeß-Erzeugung pro Client

- Prozeß-Erzeugung pro Client
  - Vorteile:
    - ◇ `fork` ist portabel (außer auf Win32)
    - ◇ Service kann Schutzkonzepte über User-ID des Clients ausnutzen.
    - ◇ Effizient für lang-dauernde Services (ftp, rlogin)
    - ◇ Transparente Ausnutzung mehrerer CPUs einer Server-Maschine

- Nachteile:
  - ◇ Vergeudung von Ressourcen wir Hauptspeicher, Prozeß-Tabelle
  - ◇ Overhead durch Kontext-Wechsel zwischen den Prozessen
  - ◇ Kommunikation/Synchronisation zwischen den Server-Prozessen schwierig
  - ◇ Behandlung der Prozeß-Terminierung (SIGCHLD) ist weder trivial noch portabel.

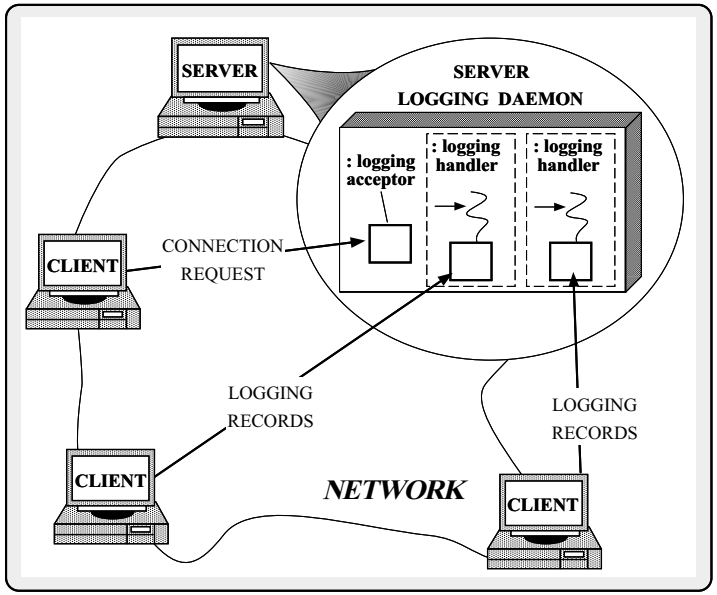

Abb. PRG–6 Server, basierend auf Thread-Erzeugung pro Client

- Thread-Erzeugung pro Client
  - Vorteile:
    - ◇ Einfacher zu programmieren (im Vergleich zu fork und SIGCHLD)
    - ◇ Effizienter: Thread-Erzeugung vs. Prozeß-Erzeugung
  - Nachteile:
    - ◇ Nicht portabel
    - ◇ Kein Schutz-Konzept zwischen den Threads verschiedener Clients

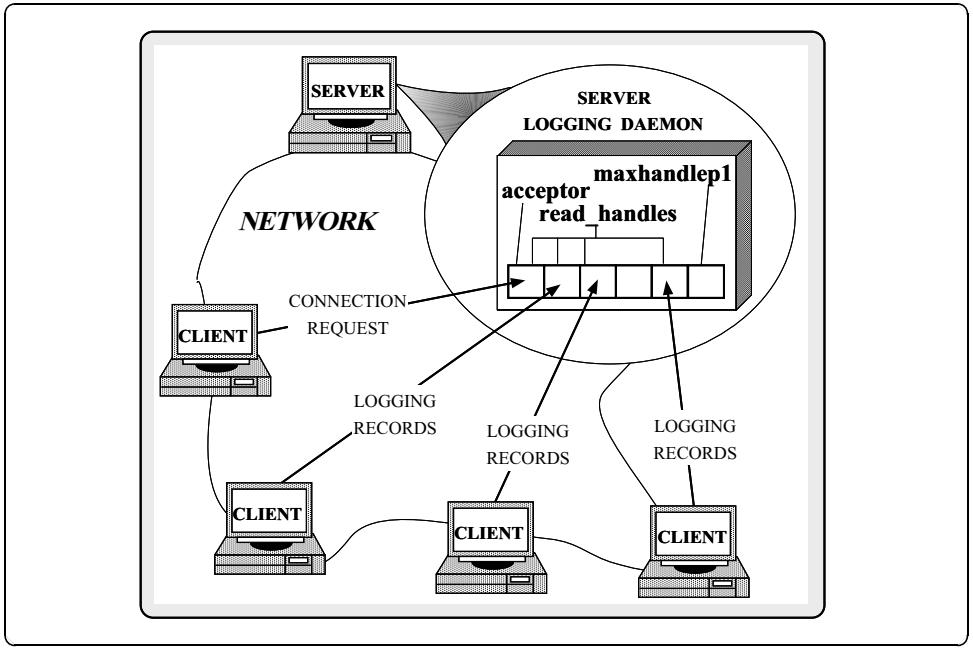

Abb. PRG–7 Server, basierend auf select

- select
  - Vorteile:
    - ◇ Effizienter als Multithreading und Prozeß-Erzeugung
      - ▷ Kein Umschalten zwischen Threads oder Prozessen
    - ◇ Keine Vergeudung von CPU-Ressourcen wie bei Polling (“busy waiting”)
  - Nachteile:
    - ◇ Kompliziertes low-level Interface
    - ◇ Verwaltung von maxhandlep1 ist diffizil bei close
    - ◇ Pro-Prozeß Limit für File-Handles vom Betriebssystem bestimmt
    - ◇ Keine Ausnutzung mehrerer CPUs einer Server-Maschine

Strategien der Thread-Erzeugung in Server-Programmen

- Single-Threaded Server
  - Vorteile:
    - ◇ Effizienz (keine Verwaltung von Threads oder Prozessen)
  - Nachteile:
    - ◇ Komplexe Programmierung (ein einziger Thread muß allen Ereignissen gerecht werden)
- Thread-per-Request

  Für jede Anforderung eines Clients wird ein neuer Thread erzeugt.
  - Vorteile:
    - ◇ Einfache Programmierung
    - ◇ Parallele Unterstützung von Multi-Threaded Clients
  - Nachteile:
    - ◇ Hoher Aufwand durch häufiges Erzeugen neuer Threads
- Thread-per-Session

  Erzeugung von Threads jeweils für die Laufzeit ("Lebensdauer") eines Clients
  - Vorteile:
    - ◇ Vermeidung des Aufwands für häufiges Erzeugen neuer Threads
  - Nachteile:
    - ◇ Abbildung von Anforderung auf den korrekten Thread ggf. schwierig
- Pool of Threads

  Ein Worker-Pool von Threads bearbeitet eintreffende Anforderungen je nach Verfügbarkeit
  - Vorteile:
    - ◇ Vermeidung des Aufwands für häufiges Erzeugen neuer Threads
  - Parallele Unterstützung von Multi-Threaded Clients
  - Nachteile:
    - ◇ Anzahl der Threads im Pool bestimmt Performance und Grad der Parallelität des Servers
    - ◇ Zusätzlicher Aufwand zur Verwaltung des Pools und zum Dispatching der Anforderungen auf die Threads.