TCP/IP – part 3
Programming with sockets

Last modification date: 08.04.2020

Client-server model

- Client-server pair implements an asymmetric relationship between two communicating processes:
  - Server: a process which is (passively) waiting at a “well-known-address” for a client request. The service offered is known to the clients as well as communication protocol
  - Client: a process which initiates communication to pass its request and possibly retrieve response
- A process can play a client role in one client-server relationship and a server in another relationship (=>multi-tier architecture)
- A process can be a provider of more than one service
- Client-server communication type:
  - Connection-oriented (stream)
  - Connectionless (datagram)
- Server can process multiple requests:
  - concurrently
  - iteratively

	Connectionless / datagram communication	Connection-oriented /stream communication
Iterative server	+	
Concurrent server		+

- A protocol of a service might require a state of communication session. Common solution:
  - State is kept by a server (e.g. FTP)
  - State is kept by a client (e.g. NFS) – the server is stateless.

What are pros and cons of the first and the second choice? Consider a crash of a client or server…

Peer-to-peer (P2P)

- P2P does not distinguish clients and servers
  - Instead all nodes are considered peers
  - May each act as client, server or both
  - Node must join P2P network
    - Registers its service with central lookup service on network (e.g. Napster) or
    - Broadcast request for service and respond to requests for service via a ***discovery protocol*** (eg. Gnutella, Freenet)

- Potential advantages over client-server:
  - Last vulnerable to computer crashes and server overloading („load balancing”)
  - Good scalability (?)
- Disadvantages:
  - Cannot satisfy high security standards
  - Unpredictable availability and quality of access
  - Possible problems with data coherence, data versions.

Service quality criteria

- Exactness of fulfilling service specs
- Response time: average, maximum, variability
- Security
- Scalability
- Reliability, availability (HA)
- Other, e.g.:
  - Open vs proprietary
  - Capable of operation in heterogeneous environment

Handling multiple communication channels

1. Blocking (default) I/O operations +
   a) Working threads – light, no overhead of local communication/memory sharing (but: synchronization needed!), crash of thread might lead to the crash of the whole process.
   b) Working sub-processes – overhead of process creation, IPC; crash of a sub-process might not influence (significantly) other sub-processes and the main.
2. Synchronous I/O multiplexing– switching of a single thread between descriptors which are ready for intended operations (see later).
3. Non-blocking I/O – advanced programming
4. Signal controlled I/O – difficult to program and not reliable (*)

.

Blocking I/O

Reading functions (`recvfrom`, `read`, `recv`) are blocking until:

- Error occurs (error code in `errno`), or when function was interrupted by a signal delivery (–1 returned, `errno==EINTR`)
- Datagram arrives (for a datagram socket).
- At least `SO_RCVLOWAT` bytes (a `receive low-water mark` (def.: 1)) are in receive buffer.); protocol option `SO_RCVLOWAT` is changeable via `setsockopt()` call.

Note:

1. Default socket settings make the reading functions block.
2. It is possible to automatically restart function which was interrupted by signal delvery; `SA_RESTART` flag setting is needed when defining signal handling (see: `man sigaction`)

Blocking I/O – cont.

Writing functions (`sendto`, `write`, `send`) are normally blocking until:

- Error occurs (error code in `errno`), or when function was interrupted by a signal delivery (–1 returned, `errno==EINTR`)
- Datagram was copied to the output queue of the channel layer of datagram socket protocol stack.
- All data was copied from the user buffer to the connection-oriented socket output buffer

`accept` function blocks until:

- It extracts the first connection request on the queue of pending connections for the listening socket, creates a new connected socket, and returns a new file descriptor referring to that socket.
- Error occurs or when function was interrupted by a signal delivery

`connect` function blocks until:

- Connection with a server is established
- Error occurred, a connection attempt was terminated or when function was interrupted by a signal delivery. In the latter case connection process is continued "in background". Connection can be checked with `select`() call (socket descriptor becomes writeable). Error can be retrieved with `getsockopt()` call (socket level option `SO_ERROR`)

Blocking I/O – cont.

`SIGALRM` signal can be used to implement time-limited operation (time-out).

<u>Example</u> (tcpudpsv.c)

Setting up `SIGALRM`

`Signal handling`

Alarm clock set (5 secs)

Attempt to pickup a connection

accept() interrupted by a signal

Connection handling

```
void ALRMhand(int sig){return;/* trivial handler */
. . .
int newsock, sockfd;
static struct sigaction sa;
sigset_t zeromask;

...
   memset(&act, 0, sizeof(struct sigaction));
   act.sa_handler = ALRMhand;
   if (-1==sigaction(SIGALRM, &act, NULL)){. . . }
   sigemptyset(&zeromask);
…
   alarm(5); /* to interrupt accept()
                     if client disconnected */
   newsock=accept(clisock,. . .);
   if (newsock==-1){
          if(errno==EINTR){/* signal occured */
                     . . .  /* accept() interrupted */
          }
   } else {/* newsock is a data socket of
                     a new connection */
   . . .
   }
   . . .

}
```

Synchronous I/O multiplexing

`select``()` function can be used to wait for a descriptor, which belongs to a specified set, to be ready for a selected class of I/O operation without blocking.

```
#include <sys/select.h>
#include <sys/time.h>
int select(int maxfd1, // highest numbered descriptor of interest +1
     fd_set *rdset, // set of descriptors to be ready for reading
     fd_set *wrset, // set of descriptors to be ready for writing
     fd_set *exset, // set of descriptors to be ready for reading OOB
   data
     const struct timeval *timeout // max. waiting time; if NULL -
                          select can block indefinitely
); // -1: error or signal interrupt; 0 - timeout;
   // >0: number of ready descriptors

FD_ZERO(fd_set *mask); // zeroes the mask
FD_SET(int bit, fd_set *mask); // sets the selected bit of the mask
FD_CLR(int bit, fd_set *mask); // clears the selected bit
FD_ISSET(int bit, fd_set *mask); // returns the selected bit
```

```
struct timeval{
   long tv_sec;
   long tv_usec;
}
```

`poll()` is a function similar to `select()` in that it can be used to wait for events (including read/writing readiness) for a set of descriptors (see man **poll**(2)).

Synchronous I/O multiplexing – cont.

Example. Use of `select` to multiplex between two descriptors `fd1` (reading) and `fd2` (writing).

```
fd_set   rdmask,   wrmask;
int fd1=..., fd2=..., numfds;
struct timeval timeout={5,0};   /* 5 second time-out */
while(1){
  FD_ZERO(&rdmask); FD_SET(fd1,&rdmask);
  FD_ZERO(&wrmask);  FD_SET(fd2,&wrmask);
  numfds=(fd1>fd2?fd1:fd2);
  if ((numfds=select(numfds+1,&rdmask,&wrmask,NULL,&timeout))<0){
               perror("select failed ");
               if(errno==EINTR) continue; else exit(1);
  }
  if (numfds==0)
        continue; /* timeout expired */
  if (FD_ISSET(fd1,&rdmask)) { /* fd1 ready ? */
     ...................
  }
  if (FD_ISSET(fd2,&wrmask)) {/* fd2 ready ? */
     ....................
  }
}/* while(1) */
```

Synchronous I/O multiplexing – cont.

Conditions for a descriptor to become **ready for reading**:

- Receiving buffer contains at least `SO_RCVLOWAT` bytes (UDP, TCP, def.:1)
- Sending end of a connection was closed (EOF is to be read)
- For a listening socket at least one connection is pending
- A socket error occurred (`so_error`, error code can be read by `getsockopt` call with option `SO_ERROR`); `read` call fails setting `error=so_error` and `so_error=0`, only if there is no input data; otherwise – data are read and `so_error` keeps its value.

Conditions for a descriptor to become **ready for writing**:

- Output buffer has at least `SO_SNDLOWAT` bytes of free space (UDP, TCP, typ. def.: 2048)
- A socket error occurred (`so_error`, error code can be read by `getsockopt` call with option `SO_ERROR`); `write` call fails setting `error=so_error` and `so_error=0`.

Conditions for a descriptor to become **ready for reading OOB data**:

- There are OOB data for a socket
- Position in the data stream points at the OOB data marker.

Synchronous I/O multiplexing – cont.

- **pselect** POSIX function for synchronous I/O multiplexing

```
#include <sys/select.h>
#include <time.h>
int pselect (int maxfd1, // highest numbered descriptor of interest +1
     fd_set *rdset, // set of descriptors to be ready for reading
     fd_set *wrset, // set of descriptors to be ready for writing
     fd_set *exset, // set of descriptors to be reade for reading OOB data
     const struct timespec *timeout // timeout; for timeout==NULL – indefinite blocking
     const sigset_t *sigmask // if not NULL – points at a signal mask set while waiting
); // -1: error or signal interrupt; 0 - timeout;
   // >0: number of ready descriptors
```

```
struct timespec{
   long tv_sec;
   long tv_nsec;
}
```

Note:

- `select()` and `pselect() differ` in precision of time-out :
- `pselect(nfds,&rmask,&wmask,&emask,&tmout,&smask);`

  is equivalent to <u>atomic</u> execution of the following functions :

```
sigset_t orgmask;
sigprocmask(SIG_SETMASK, &sigmask,&orgmask);
ready=select(nfds,&rmask,&wmask,&emask,&tmout);
sigprocmask(SIG_SETMASK,&orgmask,NULL);
```

connect() + pselect()

Example. Making connection with protection against side-effects of premature exit from `connect()`, caused by asynchronous signal handling (see Tutorial 7):

```
int connect_socket(char *name){
   struct sockaddr_un addr;
   int socketfd;
   socketfd = make_socket(name,&addr);
   if(connect(socketfd,(struct sockaddr*) &addr,SUN_LEN(&addr)) < 0){
      if(errno!=EINTR) ERR("connect");
      else {
         fd_set wfds ;
         int status;
         socklen_t size = sizeof(int);
         FD_ZERO(&wfds);FD_SET(socketfd, &wfds);
         if(TEMP_FAILURE_RETRY(select(socketfd+1,NULL,&wfds,NULL,NULL))<0)
            ERR("select");
         if(getsockopt(socketfd,SOL_SOCKET,SO_ERROR,&status,&size)<0)
            ERR("getsockopt");
         if(0!=status) ERR("connect");
      }
  }
  return socketfd;
}
```

Non-blocking I/O

Two techniques to set a descriptor to non-blocking mode

```
// First technique
int flag=1;
if (ioctl(fd, FIONBIO, &flag) == -1)
{
    perror("ioctl FIONBIO");
  .....................
}
```

```
// Second technique
int oldflag, newflag;
if ((oldflag=fcntl(fd, F_GETFL))<0)
  {
   perror("fcntl F_GETGL");
  .................
}
newflag=oldflag | FNDELAY;
if (fcntl(fd, F_SETFL, newflag)<0)
  {
   perror("fcntl F_SETFL");
  ........
}
```

When a file/socket descriptor is set into a non-blocking mode the functions `accept, read, recv, recvfrom, send, sendto, write` do not block. If they cannot perform their normal activity they return -1 after setting `errno` to `EWOULDBLOCK` or `EAGAIN.` Functions calls are usually retried, frequently after checking descriptor readiness with `select()` call.

Consequences of non-blocking I/O mode:

- Very good utilization of channel bandwidth
- Complication of I/O buffer management, because of partial reads/writes and a need for descriptor polling (e.g. with **select()/pselect()** function call)

Non-blocking I/O

`connect` function can return -1 and set `EINPROGRESS`. Processing is then done asynchronously. No `connect` function call retries are needed. Successful connection can be detected by `select` function call (descriptor is ready for writing). Error can be checked with `getsockopt` function call (socket option: `SOCKERR`).

`accept` function can block the caller, when a descriptor (in blocking mode) is found by `select` as ready for reading, but connection is terminated by a client before `accept` call is actually made. To prevent the blocking the listening descriptor should be set to nonblocking mode and when `select` finds the descriptor ready for reading the following code should be executed (see Tutorial 7):

```
int add_new_client(int sfd){ /* sfd: listening nonblocking mode socket */
   int nfd;
   if((nfd=TEMP_FAILURE_RETRY(accept(sfd,NULL,NULL)))<0) {
      if(EAGAIN==errno||EWOULDBLOCK==errno) return -1;
      ERR("accept");
   }
   return nfd;
}
```

SIGIO and I/O(*)

It is possible that a kernel generates `SIGIO` upon readiness of a descriptor. Signal handler has to recognize the reason for signal delivery (and so the file descriptor) and subsequently initiate I/O operation.

Example. Setting asynchronous notification mode for descriptor `fd`

```
//// First technique ////
int         flag = -getpid();
if (ioctl(fd, SIOCSPGRP, &flag) == -1)
   {
   perror("ioctl SIOCSOGRP");
   ,,,,,,,,,,,,,,,
}
flag=1;
if (ioctl(fd, FIOASYNC, &flag) == -1) {
   perror("ioctl FIOASYNC");
   ...............
}
```

```
//// Second technique ////

if (fcntl(fd, F_SETOWN, getpid())<0){
   perror("fcntl F_SETOWN");
   .................
}
if (fcntl(fd, F_SETFL, O_ASYNC)<0) {
   perror("fcntl F_SETFL O_ASYNC");
   .............
}
```

Note:

- Before setting the asynchronous notification it is necessary to setup `SIGIO` signal handling.
- For TCP sockets the notification is almost useless, because signals can be too frequent.
- Signals can be lost (if signal handling is not fast enough).

Running servers

- TCP/IP server processes typically run as daemons.
- Daemon – a background process which cannot be controlled from any terminal.
- In Linux **daemon()**, function can be used to make a process a deamon,
- Traditional methods to start servers
  - "From shell scripts, which are run upon system boot (e.g. `/etc/rc`, `/etc/rc.local`)
  - Via a super-server, which starts at boot-up and monitor client requests.
  - Via a system deamon, like `cron` (for configuration see `crontab(5)`).
  - From user terminal (for testing purposes)

Remarks

- Since deamons cannot use terminals so they output messages via system log facilitis, e.g using `syslog` daemon (see `syslog(3C),syslog.conf(4)`.

- For some time **systemd** software is gaining ground (see e.g. **man systemd(1), systemd.unit(5)**). The software implements, e.g. functionality of the traditional UNIX daemon **init** (PID=1) and a part of functionality implemented in UNIX daemon processes. In effect writing daemons is easier and daemon management is more powerful.