Programmation Systèmes
Cours 9 — UNIX Domain Sockets

Stefano Zacchiroli
zack@pps.univ-paris-diderot.fr

Laboratoire PPS, Université Paris Diderot

2013–2014

URL http://upsilon.cc/zack/teaching/1314/progsyst/
Copyright © 2013 Stefano Zacchiroli
License Creative Commons Attribution-ShareAlike 4.0 International License
http://creativecommons.org/licenses/by-sa/4.0/deed.en_US
Outline

1. Sockets
2. Stream sockets
3. UNIX domain sockets
4. Datagram sockets
1. Sockets
2. Stream sockets
3. UNIX domain sockets
4. Datagram sockets
Sockets are IPC objects that allow to exchange data between processes running:
- either on the same machine (*host*), or
- on different ones over a network.

The UNIX socket API first appeared in 1983 with BSD 4.2. It has been finally standardized for the first time in POSIX.1g (2000), but has been ubiquitous to every UNIX implementation since the 80s.

**Disclaimer**

The socket API is best discussed in a network programming course, which this one is *not*. We will only address enough general socket concepts to describe how to use a specific socket family: **UNIX domain sockets**.
Client-server setup

Let’s consider a typical client-server application scenario — no matter if they are located on the same or different hosts.

Sockets are used as follows:

- **each application**: create a socket
  - idea: communication between the two applications will flow through an imaginary “pipe” that will connect the two sockets together
- **server**: bind its socket to a **well-known address**
  - we have done the same to set up rendez-vous points for other IPC objects, e.g. FIFOs
- **client**: locate server socket (via its well-known address) and “initiate communication”\(^1\) with the server

\(^1\)various kinds of communication are possible, so we will refine this later
Sockets are created using the `socket` syscall which returns a file descriptor to be used for further operations on the underlying socket:

```c
fd = socket(domain, type, protocol);
```

Each triple \( \langle \text{domain}, \text{type}, \text{protocol} \rangle \) identifies a different “species” of sockets.

For our purposes protocol will always be 0, so we don’t discuss it further.
Communication domains

Each socket exists within a communication domain.

Each communication domain determines:

- **how to identify a socket**, that is the syntax and semantics of socket well-known addresses
- **the communication range**, e.g. whether data flowing through the socket span single or multiple hosts

Popular socket communication domains are:

- **UNIX** communication within the same machine, using pathnames as addresses
- **IPv4** communication across hosts, using IPv4 addresses (e.g. `173.194.40.128`)
- **IPv6** communication across hosts, using IPv6 addresses (e.g. `2a00:1450:4007:808::1007`
## Communication domains — details

<table>
<thead>
<tr>
<th>domain</th>
<th>range</th>
<th>transport</th>
<th>address format</th>
<th>address C struct</th>
</tr>
</thead>
<tbody>
<tr>
<td>AF_UNIX</td>
<td>same host</td>
<td>kernel</td>
<td>pathname</td>
<td>sockaddr_un</td>
</tr>
<tr>
<td>AF_INET</td>
<td>any host w/ IPv4 connectivity</td>
<td>IPv4 stack</td>
<td>32-bit IPv4 address + 16-bit port number</td>
<td>sockaddr_in</td>
</tr>
<tr>
<td>AF_INET6</td>
<td>any host w/ IPv6 connectivity</td>
<td>IPv6 stack</td>
<td>128-bit IPv6 address + 16-bit port number</td>
<td>sockaddr_in6</td>
</tr>
</tbody>
</table>

**fd = socket(domain, type, protocol);**

---

\(^2\) value for the first argument of the socket syscall
fd = socket(domain, type, protocol);

Within each socket domain you will find multiple socket types, which offer different IPC features:

<table>
<thead>
<tr>
<th>feature</th>
<th>socket type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SOCK_STREAM</td>
</tr>
<tr>
<td>reliable delivery</td>
<td>yes</td>
</tr>
<tr>
<td>message boundaries</td>
<td>no</td>
</tr>
<tr>
<td>connection-oriented</td>
<td>yes</td>
</tr>
</tbody>
</table>
Stream sockets (SOCK_STREAM)

Stream sockets provide communication channels which are:

- **byte-stream**: there is no concept of message boundaries, communication happens as a continuous stream of bytes
- **reliable**: either data transmitted arrive at destination, or the sender gets an error
- **bidirectional**: between two sockets, data can be transmitted in either direction
- **connection-oriented**: sockets operate in **connected pairs**, each connected pair of sockets denotes a communication context, isolated from other pairs
  - a peer socket is the other end of a given socket in a connection
  - the peer address is its address

**Intuition**

Stream sockets are like pipes, but also permit (in the Internet domains) communication across the network.
Datagram sockets (SOCK_DGRAM)

Datagram sockets provide communication channels which are:

- message-oriented: data is exchanged at the granularity of messages that peers send to one another; message boundaries are preserved and need not to be created/recognized by applications

- non-reliable: messages can get lost. Also:
  - messages can arrive out of order
  - messages can be duplicated and arrive multiple times

It is up to applications to detect these scenarios and react (e.g. by re-sending messages after a timeout, add sequence number, etc.).

- connection-less: sockets do not need to be connected in pairs to be used; you can send a message to, or receive a message from a socket without connecting to it beforehand
In the **Internet domains** (AF_INET and AF_INET6):

- socket communications happen over the **IP** protocol, in its IPv4 and IPv6 variants (Internet layer)
- stream sockets use the **TCP** protocol (transport layer)
- datagram sockets use the **UDP** protocol (transport layer)

You’ll see all this in the network programming course...
$ netstat -txun

Active Internet connections (w/o servers)

<table>
<thead>
<tr>
<th>Proto</th>
<th>Recv-Q</th>
<th>Send-Q</th>
<th>Local Address</th>
<th>Foreign Address</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>tcp</td>
<td>1</td>
<td>1</td>
<td>128.93.60.82:53161</td>
<td>98.137.200.255:80</td>
<td>LAST_ACK</td>
</tr>
<tr>
<td>tcp</td>
<td>0</td>
<td>0</td>
<td>10.19.0.6:54709</td>
<td>10.19.0.1:2777</td>
<td>ESTABLISHED</td>
</tr>
<tr>
<td>tcp</td>
<td>0</td>
<td>0</td>
<td>128.93.60.82:53366</td>
<td>98.137.200.255:80</td>
<td>ESTABLISHED</td>
</tr>
<tr>
<td>tcp</td>
<td>0</td>
<td>0</td>
<td>10.19.0.6:46368</td>
<td>10.19.0.1:2778</td>
<td>ESTABLISHED</td>
</tr>
<tr>
<td>tcp</td>
<td>0</td>
<td>0</td>
<td>128.93.60.82:47218</td>
<td>74.125.132.125:5222</td>
<td>ESTABLISHED</td>
</tr>
<tr>
<td>tcp6</td>
<td>1</td>
<td>0</td>
<td>::1:51113</td>
<td>::1:631</td>
<td>CLOSE_WAIT</td>
</tr>
<tr>
<td>udp</td>
<td>0</td>
<td>0</td>
<td>127.0.0.1:33704</td>
<td>127.0.0.1:33704</td>
<td>ESTABLISHED</td>
</tr>
</tbody>
</table>

Active UNIX domain sockets (w/o servers)

<table>
<thead>
<tr>
<th>Proto</th>
<th>RefCnt</th>
<th>Flags</th>
<th>Type</th>
<th>State</th>
<th>I-Node</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>unix</td>
<td>2</td>
<td>[]</td>
<td>DGRAM</td>
<td></td>
<td>23863</td>
<td>/var/spool/postfix/dev/log</td>
</tr>
<tr>
<td>unix</td>
<td>2</td>
<td>[]</td>
<td>DGRAM</td>
<td></td>
<td>1378</td>
<td>/run/systemd/journal/syslog</td>
</tr>
<tr>
<td>unix</td>
<td>2</td>
<td>[]</td>
<td>DGRAM</td>
<td></td>
<td>1382</td>
<td>/run/systemd/shutdown</td>
</tr>
<tr>
<td>unix</td>
<td>2</td>
<td>[]</td>
<td>DGRAM</td>
<td></td>
<td>4744</td>
<td>@/org/freedesktop/systemd1/notify</td>
</tr>
<tr>
<td>unix</td>
<td>5</td>
<td>[]</td>
<td>DGRAM</td>
<td></td>
<td>1390</td>
<td>/run/systemd/journal/socket</td>
</tr>
<tr>
<td>unix</td>
<td>28</td>
<td>[]</td>
<td>DGRAM</td>
<td></td>
<td>1392</td>
<td>/dev/log</td>
</tr>
<tr>
<td>unix</td>
<td>3</td>
<td>[]</td>
<td>STREAM</td>
<td>CONNECTED</td>
<td>138266</td>
<td></td>
</tr>
<tr>
<td>unix</td>
<td>2</td>
<td>[]</td>
<td>STREAM</td>
<td>CONNECTED</td>
<td>79772</td>
<td></td>
</tr>
<tr>
<td>unix</td>
<td>3</td>
<td>[]</td>
<td>STREAM</td>
<td>CONNECTED</td>
<td>30935</td>
<td></td>
</tr>
<tr>
<td>unix</td>
<td>3</td>
<td>[]</td>
<td>STREAM</td>
<td>CONNECTED</td>
<td>23037</td>
<td></td>
</tr>
<tr>
<td>unix</td>
<td>3</td>
<td>[]</td>
<td>STREAM</td>
<td>CONNECTED</td>
<td>416650</td>
<td></td>
</tr>
<tr>
<td>unix</td>
<td>3</td>
<td>[]</td>
<td>SEQPACKET</td>
<td>CONNECTED</td>
<td>135740</td>
<td></td>
</tr>
<tr>
<td>unix</td>
<td>3</td>
<td>[]</td>
<td>STREAM</td>
<td>CONNECTED</td>
<td>26655</td>
<td>/run/systemd/journal/stdout</td>
</tr>
<tr>
<td>unix</td>
<td>2</td>
<td>[]</td>
<td>DGRAM</td>
<td></td>
<td>22969</td>
<td></td>
</tr>
<tr>
<td>unix</td>
<td>3</td>
<td>[]</td>
<td>STREAM</td>
<td>CONNECTED</td>
<td>29256</td>
<td>@/tmp/dbus-tHnZVgCvqF</td>
</tr>
<tr>
<td>unix</td>
<td>3</td>
<td>[]</td>
<td>STREAM</td>
<td>CONNECTED</td>
<td>91045</td>
<td>@/tmp/dbus-tHnZVgCvqF</td>
</tr>
</tbody>
</table>

...
Socket creation can be requested using socket:

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

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

Returns: file descriptor on success, -1 on error
```

As we have seen, the 3 arguments specify the “species” of socket you want to create:

- **domain**: AF_UNIX, AF_INET, AF_INET6
- **type**: SOCK_STREAM, SOCK_DGRAM
- **protocol**: always 0 for our purposes

The file descriptor returned upon success is used to further reference the socket, for both communication and setup purposes.

---

3 one case in which it is non-0 is when using raw sockets
Binding sockets to a well-known address

To allow connections from other, we need to bind sockets to well-known addresses using `bind`:

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

int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

Returns: 0 on success, -1 on error

- `sockfd` references the `socket` we want to bind
- `addr` and `addrlen` are, respectively, the length and the structure containing the well-known address we want to bind the socket to

The actual type of the `addr` structure depends on the socket domain...
Generic socket address structure

We have seen that the address format varies with the domain:

- UNIX domain uses pathnames
- Internet domains use IP addresses

But bind is a generic system call that can bind sockets in any domain! Enter `struct sockaddr`:

```c
struct sockaddr {
    sa_family_t sa_family; /* address family (AF_*) */
    char sa_data[14]; /* socket address (size varies with the socket domain) */
}
```

- each socket domain has its own variant of sockaddr
- you will fill the domain-specific struct
- and cast it to `struct sockaddr` before passing it to `bind`
- `bind` will use `sa_family` to determine how to use `sa_data`
Outline

1. Sockets
2. Stream sockets
3. UNIX domain sockets
4. Datagram sockets
The phone analogy for stream sockets

Stream socket / phone analogy

To communicate one application—which we call “client”—must call the other—the “server”—over the phone. Once the connection is established, each peer can talk to the other for the duration of the phone call.

- both: `socket()` → install a phone
- server: `bind()` → get a phone number
- server: `listen()` → turn on the phone, so that it can ring
- client: `connect()` → turns on the phone and call the “server”, using its number
- server: `accept()` → pick up the phone when it rings (or wait by the phone if it’s not ringing yet)
Stream socket syscalls — overview

Terminology

“Server” and “client” are ambiguous terms. We speak more precisely of passive and active sockets.

- sockets are created active; listen() makes them passive
- connect() performs an active open
- accept() performs a passive open
Willingness to accept connections

listen turns an active socket into a passive one, allowing him to accept incoming connections (i.e. performing passive opens):

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

int listen(int sockfd, int backlog);

Returns: 0 on success, -1 on error
```

- sockfd references the socket we want to affect
- backlog specifies the maximum number of pending connections that the passive socket will keep
Pending connections

- active opens may be performed before the matching passive ones
- not yet accept-ed connections are called pending
- they may increase or decrease over time, depending on the serving time
- with pending < backlog, connect succeeds immediately
- with pending >= backlog, connect blocks waiting for an accept

TLPI, Fig. 56-2
Accepting connections

You can accept connections (i.e. perform a passive open) with:

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

int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);

Returns: file descriptor on success, -1 on error
```

If the corresponding active open hasn’t been performed yet, `accept` blocks waiting for it. When the active open happens—or if it has already happened—`accept` returns a new socket connected to the peer socket. The original socket remains available and can be used to accept other connections.

`addr/addrlen` are value-result arguments which will be filled with the address of the peer socket. Pass NULL if not interested.

- note: differently from other IPC mechanisms, we might know “who” is our peer
To close the puzzle, you connect (i.e. perform an active open) with:

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

int connect(int sockfd, struct sockaddr *addr, socklen_t addrlen);

Returns: 0 on success, -1 on error
```

- `sockfd` is *your own socket*, to be used as your endpoint of the connection
- `addr/addrlen` specify the *well-known address of the peer* you want to connect to, and are given in the same format of `bind` parameters
Communicating via stream sockets

Once a connection between two peer socket is established, communication happens via read/write on the corresponding file descriptors:

\[ \text{Application A} \quad \text{Kernel} \quad \text{Application B} \]

\[ \text{sockfd} \quad \text{buffer} \quad \text{buffer} \quad \text{sockfd} \]

`close` on one end will have the same effects of closing one end of a pipe:

- reading from the other end will return EOF
- writing to the other end will fail with EPIPE error and send SIGPIPE to the writing process
Outline

1. Sockets
2. Stream sockets
3. UNIX domain sockets
4. Datagram sockets
Socket addresses in the UNIX domain

We now want to give an example of stream sockets. To do so, we can longer remain in the abstract of general sockets, but we need to pick a domain. We pick the UNIX domain.

In the UNIX domain, addresses are pathnames. The corresponding C structure is `sockaddr_un`:

```c
struct sockaddr_un {
    sa_family_t sun_family; /* = AF_UNIX */
    char sun_path[108]; /* socket pathname, NULL-terminated */
}
```

The field `sun_path` contains a regular pathname, pointing to a special file of type socket (≠ pipe) which will be created at bind time.

During communication the file will have no content, it is used only as a rendez-vous point between processes.
const char *SOCK_PATH = "/tmp/srv_socket";
int srv_fd;
struct sockaddr_un addr;

srv_fd = socket(AF_UNIX, SOCK_STREAM, 0);
if (srv_fd < 0)
    err_sys("socket error");

memset(&addr, 0, sizeof(struct sockaddr_un));
    /* ensure that all fields, including non-standard ones, are initialized to 0 */
addr.sun_family = AF_UNIX;
strncpy(addr.sun_path, SOCK_PATH, sizeof(addr.sun_path) - 1);
    /* we copy one byte less, ensuring a trailing 0 exists */

if (bind(srv_fd, (struct sockaddr *)&addr,
    sizeof(struct sockaddr_un)) < 0)
    err_sys("bind error");
Binding UNIX domain socket — caveats

- the actual filesystem entry is created at **bind time**
  - if the file already exists, **bind** will fail
  - it’s up to you to remove stale sockets as needed
- **ownership/permissions** on the file are determined as usual (effective user id, umask, etc.)
  - to connect to a socket you need **write permission** on the corresponding file
- `stat().st_mode == S_IFSOCK` and `ls` shows:
  ```bash
  /var/run/systemd$ ls -lF shutdownd
  srw------ 1 root root 0 dic 9 19:34 shutdownd=
  ```
- you can’t **open()** a UNIX domain socket, you must **connect()** to it
To experiment with stream sockets in the UNIX domain we will write a client-server **echo application** where:

- the client connects to the server and transfers its entire **standard input** to it
- the server accepts a connection, and transfers all the data coming from it to **standard output**
- the server is iterative: it processes one connection at a time, reading all of its data (potentially infinite) before processing other connections
#include <sys/un.h>
#include <sys/socket.h>
#include <unistd.h>
#include "helpers.h"

#define SRV_SOCK_PATH  "/tmp/ux_socket"

#define BUFFSIZE 1024
#define SRV_BACKLOG 100

/* end of stream−proto.h */
```c
#include "stream-PROTO.h"

int main(int argc, char **argv) {
  struct sockaddr_un addr;
  int srv_fd, cli_fd;
  ssize_t bytes;
  char buf[BUFFSIZE];

  if ((srv_fd = socket(AF_UNIX, SOCK_STREAM, 0)) < 0)
    err_sys("socket error");

  memset(&addr, 0, sizeof(struct sockaddr_un));
  addr.sun_family = AF_UNIX;
  strncpy(addr.sun_path, SRV_SOCK_PATH,
          sizeof(addr.sun_path) - 1);
  if (access(addr.sun_path, F_OK) == 0)
    unlink(addr.sun_path);
  if (bind(srv_fd, (struct sockaddr *)&addr,
           sizeof(struct sockaddr_un)) < 0)
    err_sys("bind error");
```
if (listen(srv_fd, SRV_BACKLOG) < 0)  
err_sys("listen error");

for (;;) {
    if ((cli_fd = accept(srv_fd, NULL, NULL)) < 0)  
        err_sys("accept error");

    while ((bytes = read(cli_fd, buf, BUFFSIZE)) > 0)
        if (write(STDOUT_FILENO, buf, bytes) != bytes)  
            err_sys("write error");

    if (bytes < 0)  
        err_sys("read error");

    if (close(cli_fd) < 0)  
        err_sys("close error");
}

/* end of stream-server.c */
#include "stream−proto.h"

int main(int argc, char **argv) {
    struct sockaddr_un addr;
    int srv_fd;
    ssize_t bytes;
    char buf[BUFSIZE];

    if ((srv_fd = socket(AF_UNIX, SOCK_STREAM, 0)) < 0)
        err_sys("socket error");

    memset(&addr, 0, sizeof(struct sockaddr_un));
    addr.sun_family = AF_UNIX;
    strncpy(addr.sun_path, SRV_SOCK_PATH, sizeof(addr.sun_path) − 1);
    if (connect(srv_fd, (struct sockaddr *)&addr,
                sizeof(struct sockaddr_un)) < 0)
        err_sys("connect error");
while((bytes = read(STDIN_FILENO, buf, BUFFSIZE)) > 0) {
    if (write(srv_fd, buf, bytes) != bytes)
        err_sys("write error");
    if (bytes < 0)
        err_sys("read error");
    exit(EXIT_SUCCESS);
}

/* end of stream−client.c */
Client-server stream socket example

Demo

Notes:

- the server accepts multiple connections, iteratively
- we can’t directly open its socket (e.g. using shell redirections)
Outline

1. Sockets
2. Stream sockets
3. UNIX domain sockets
4. Datagram sockets
The mail analogy for datagram sockets

Datagram socket / mail analogy

To communicate applications send (snail) mail messages to their peer mailboxes.

- both: `socket()` → installing a mailbox
- both:⁴ `bind()` → get a postal address
- peer A: `sendto()` → send a letter to peer B, writing to her postal address
- peer B: `recvfrom()` → check mailbox to see if a letter has arrived, waiting for it if it’s not the case
  - each letter is marked with the sender address, so peer B can write back to peer A even if her address is not public

As it happens with the postal system letters can be reordered during delivery and might not arrive. Additionally, with datagram sockets “letters” can be duplicated.

⁴whether you need bind to `receive` messages depends on the domain
Datagram socket syscalls — overview

Server
- socket()
- bind()
- recvfrom()
- sendto()
- close()

Client
- socket()
- sendto()
- recvfrom()
- close()

(Possibly multiple) data transfers in either direction

TLPI, Fig. 56-2
Sending datagrams

The sendto syscall is used to send a single datagram to a peer:

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

ssize_t sendto(int sockfd, void *buffer, size_t length, int flags, const struct sockaddr *dest_addr, socklen_t addrlen);

Returns: bytes sent on success, -1 on error
```

- the return value and the first 3 arguments are as in write
- flags can be specified to request socket-specific features
- dest_addr/addrlen specify the destination address
Receiving datagrams

The recvfrom is used to receive a single datagram from a peer:

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

ssize_t recvfrom(int sockfd, void *buffer, size_t length, int flags,
                  struct sockaddr *src_addr, socklen_t *addrlen);

Returns: bytes received on success, 0 on EOF, -1 on error
```

- the return value and the first 3 arguments are as in read
  - note: recvfrom always fetch exactly 1 datagram, regardless of length; if length it's too short the message will be truncated
- flags are as in sendto
- dest_addr/addrlen are value-result arguments that will be filled with the sender address; specify NULL if not interested

If no datagram is available yet, recvfrom blocks waiting for one.
UNIX domain datagram sockets

Whereas *in general* datagram sockets are not reliable, datagram sockets in the UNIX domain are *reliable*: all messages are either delivered or reported as missing to the sender, non-reordered, non-duplicated.

To be able to receive datagrams (e.g. replies from a server), you should name client sockets using *bind*.

To be able to send datagrams you need *write permission* on the corresponding file.

On Linux you can send quite large datagrams (e.g. 200 KB, see `/proc/sys/net/core/wmem_default` and the `socket(7)` manpage). On other UNIX you find limits as low as 2048 bytes.
To experiment with datagram sockets in the UNIX domain we will write a client/server application where:

- the client takes a number of arguments on its command line and send them to the server using separate datagrams
- for each datagram received, the server converts it to uppercase and send it back to the client
- the client prints server replies to standard output

For this to work we will need to bind all involved sockets to pathnames.
Client-server datagram socket example — protocol

```c
#include <ctype.h>
#include <sys/un.h>
#include <sys/socket.h>
#include <unistd.h>
#include "helpers.h"

#define SRV_SOCK_PATH " /tmp/uc_srv_socket "
#define CLI_SOCK_PATH " /tmp/uc_cli_socket.%ld "
#define MSG_LEN 10

/* end of uc−proto.h, based on TLPI Listing 57−5,
   Copyright (C) Michael Kerrisk, 2010. License: GNU AGPL−3+ */
```
#include "uc-proto.h"

int main(int argc, char *argv[]) {
    struct sockaddr_un srv_addr, cli_addr;
    int srv_fd, i;
    ssize_t bytes;
    socklen_t len;
    char buf[MSG_LEN];

    if ((srv_fd = socket(AF_UNIX, SOCK_DGRAM, 0)) < 0)
        err_sys("socket error");

    memset(&srv_addr, 0, sizeof(struct sockaddr_un));
    srv_addr.sun_family = AF_UNIX;
    strncpy(srv_addr.sun_path, SRV_SOCK_PATH, sizeof(srv_addr.sun_path) - 1);
    if (access(srv_addr.sun_path, F_OK) == 0)
        unlink(srv_addr.sun_path);
    if (bind(srv_fd, (struct sockaddr *) &srv_addr,
             sizeof(struct sockaddr_un)) < 0)
        err_sys("bind error");
for (; ; ) {
    len = sizeof(struct sockaddr_un);
    if ((bytes = recvfrom(srv_fd, buf, MSG_LEN, 0,
                              (struct sockaddr *) &cli_addr, &len)) < 1)
        err_sys("recvfrom error");
    printf("server received %ld bytes from %s\n",
           (long) bytes, cli_addr.sun_path);
    for (i = 0; i < bytes; i++)
        buf[i] = toupper((unsigned char) buf[i]);
    if (sendto(srv_fd, buf, bytes, 0,
                  (struct sockaddr *) &cli_addr, len) != bytes)
        err_sys("sendto error");
}

/* end of uc−server.c, based on TLPI Listing 57−6,
   Copyright (C) Michael Kerrisk, 2010. License: GNU AGPL−3+ */
Client-server datagram socket example — client

```c
#include "uc-proto.h"

int main(int argc, char *argv[]) {
    struct sockaddr_un srv_addr, cli_addr;
    int srv_fd, i;
    size_t len;
    ssize_t bytes;
    char resp[MSG_LEN];

    if (argc < 2)
        err_quit("Usage: uc-client MSG...");

    if ((srv_fd = socket(AF_UNIX, SOCK_DGRAM, 0)) < 0)
        err_sys("socket error");
    memset(&cli_addr, 0, sizeof(struct sockaddr_un));
    cli_addr.sun_family = AF_UNIX;
    snprintf(cli_addr.sun_path, sizeof(cli_addr.sun_path),
              CLI_SOCKET_PATH, (long) getpid());
    if (bind(srv_fd, (struct sockaddr *)&cli_addr,
             sizeof(struct sockaddr_un)) == -1)
        err_sys("bind error");
```
memset(&srv_addr, 0, sizeof(struct sockaddr_un));
srv_addr.sun_family = AF_UNIX;
strncpy(srv_addr.sun_path, SRV_SOCK_PATH,
        sizeof(srv_addr.sun_path) − 1);
for (i = 1; i < argc; i++) {
    len = strlen(argv[i]);

    if (sendto(srv_fd, argv[i], len, 0,
                   (struct sockaddr *)&srv_addr,
                   sizeof(struct sockaddr_un)) != len)
        err_sys("sendto error");
    if ((bytes = recvfrom(srv_fd, resp, MSG_LEN,
                           0, NULL, NULL)) < 0)
        err_sys("recvfrom error");
    printf("response %d: %. *s\n", i, (int) bytes, resp);
}
unlink(cli_addr.sun_path);
exit(EXIT_SUCCESS);

/* end of uc−client.c, based on TLPI Listing 57−7, 
   Copyright (C) Michael Kerrisk, 2010. License: GNU AGPL−3+ */
Notes:

- the server is persistent and processes one datagram at a time, no matter the client process, i.e. there is no notion of connection
- messages larger than 10 bytes are silently truncated