University of Stuttgart
Institute of Parallel and Distributed Systems

Distributed Systems

Fachpraktikum / Lab-Course

Software-Defined and Time-Sensitive Networking

Tutorial: Networking Basics
Part 2: Sockets

Frank Dürr

Summer Term 2023

IPVS

Agenda

- Introduction to socket API
- Socket types
- Stream socket: client and server
- Datagram socket: client and server
- Protocol-agnostic socket programming

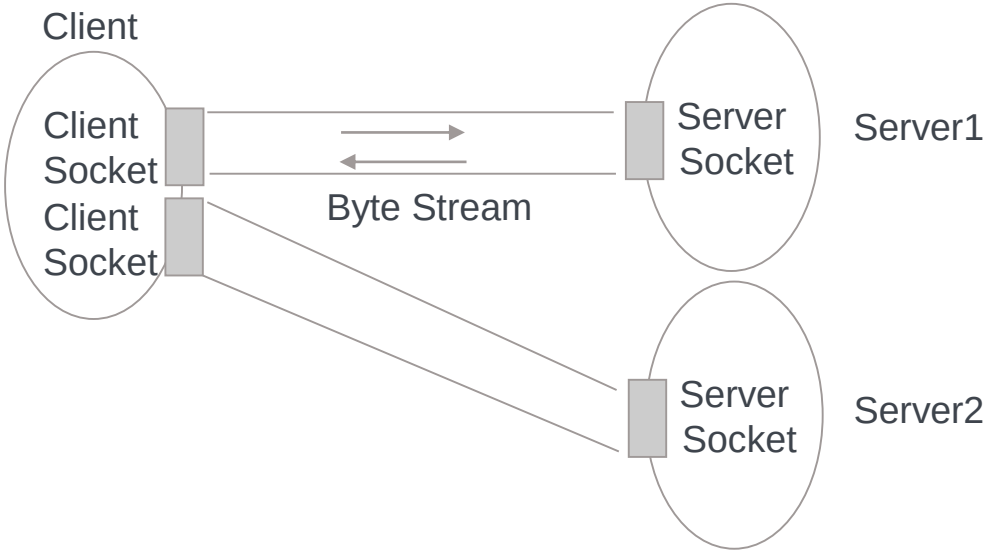
Sockets

- Application programming interface (API) for network communication
- Interface between application and network stack of operating system
  - At transport layer (mostly used)
  - At network layer or data (raw sockets)
  - At data link layer (packet sockets)
- Originated in BSD (Berkeley sockets), now POSIX standard, very similar implementation for Windows (Winsock API)
- Designed for Internet communication with TCP/UDP IP, but in general protocol-independent

Socket Types

Stream (SOCK_STREAM)

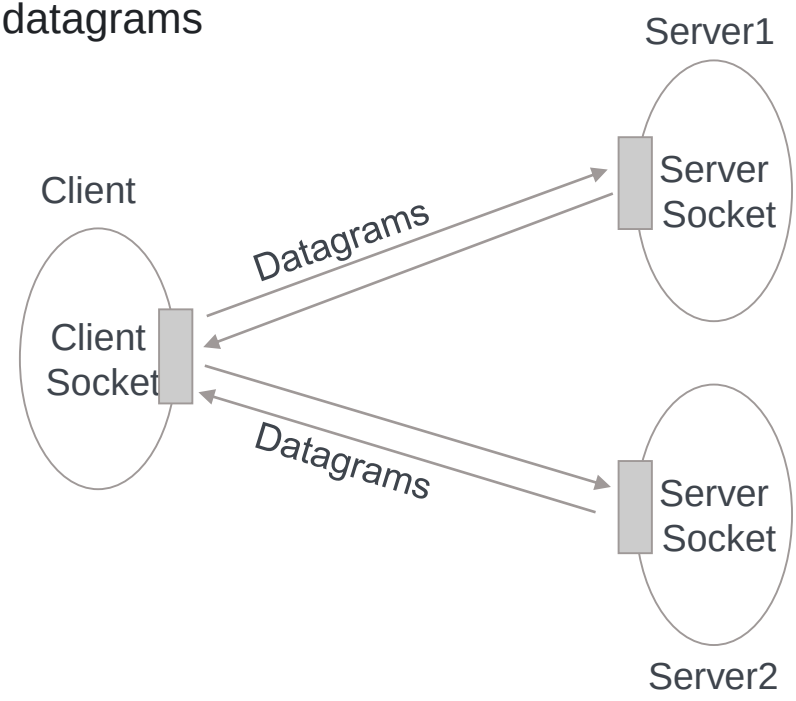
- Connection-oriented communication
- Reliable
- Sequenced (order-preserving)
- Bidirectional
- Byte stream



Socket Types

Datagram (SOCK_DGRAM)

- Connection-less communication
- Unreliable
- Fixed maximum length datagrams



Socket Types

Other Kinds of Sockets

- SOCK_SEQPACKET
  - Reliable, connection-oriented, fixed maximum-length datagrams with sequencing
- SOCK_RDM
  - Reliable datagrams without ordering
- SOCK_RAW
  - Raw network layer packets
  - Raw device-driver packets (previous SOCK_PACKET)

Stream Server Socket

1. `socket()`: create socket
2. `bind()`: bind socket to address (IP/port)
3. `listen()`: socket ready to receive connection requests from clients (queue for incoming requests)
4. `accept()`: accept connection request;
   creates new socket connected to client;
   original socket listens to further conn. requests
5. `read()/write()`: read/write bytes from/to client
   `send()/recv()`
6. `shutdown()`: (optional) prevent sending in one/both directions
   `close()`: close socket (destroys socket)

Stream Client Socket

1. `socket()`: create socket
2. `connect()`: connect client to server socket;
   operating system choses client address (IP/port)
   according to routing table and next free port
3. `read()/write()`: read/write bytes from/to client
   `send()/recv()`
4. `shutdown()`: (optional) prevent sending in one/both directions
   `close()`: close socket (destroys socket)

Datagram Server Socket

1. `socket()`: create socket
2. `bind()`: bind socket to address (IP/port)
3. `recvfrom()`: receive message
   `sendto()`: send message
4. `close()`: close socket (destroys socket)

Datagram Client Socket

1. `socket()`: create socket
2. `bind()`: bind socket to address (IP/port)
3. `recvfrom()`: receive message
   `sendto()`: send message
4. `close()`: close socket (destroys socket)

Protocol-Agnostic Socket Programming

- Avoid writing code that includes protocol details!
  - Avoid using AF_INET, AF_INET6, IPPROTO_TCP, etc. in your code!
  - Your application might not work anymore with a new (different) protocol
- Be **protocol-agnostic** and use **`getaddrinfo()`**
  - You specify what you need, e.g.:
    - Connection-oriented byte stream (SOCK_STREAM) for client
  - Operating system tells you what protocols you can try, e.g.:
    - IPv4 + TCP
    - IPv6 + TCP

Protocol-Agnostic Socket Programming

getaddrinfo()

```
int getaddrinfo(const char *node,
                const char *service,
                const struct addrinfo *hints,
                struct addrinfo **res);
```

- `node`: host name or IP-Adresse (as string)
- `service`: service name or port number (as string)
- `hints`: specification of requirements for socket
- `res`: linked list of protocols and addresses to try

`freeaddrinfo()` on `res`, when you are done

Protocol-Agnostic Socket Programming

getaddrinfo()

```
struct addrinfo {
  int ai_flags;    // 0=client; AI_PASSIVE=server
  int ai_family;   // AF_UNSPEC, AF_INET, AF_INET6
  int ai_socktype; // SOCK_STREAM, SOCK_DGRAM
  int ai_protocol; // 0 (all transport protocols)
                   // or IPPROTO_TCP, ...
  socklen_t ai_addrlen;
  struct sockaddr *ai_addr;
  char *ai_canonname;
  struct addrinfo *ai_next;
};
```

Protocol-Agnostic Socket Programming

Example: Stream Server

```
struct addrinfo hints = {
     .ai_flags = AI_PASSIVE,      // server
     .ai_family = AF_UNSPEC,      // don't care
     .ai_protocol = 0,            // don't care
     .ai_socktype = SOCK_STREAM   // reliable byte-stream
};
struct addrinfo *res;
const char *name = "www.foobar.de";
const char *service = "http"; // cf. /etc/services
getaddrinfo(name, service, &hints, &res);
```

Protocol-Agnostic Socket Programming

Example: Stream Server

```
// Try entries in list until one works
struct addrinfo *ai;
for (ai = res; ai != NULL; ai = ai->ai_next) {
  // Create socket with parameters from ai.
  // If failure, try next entry.
  // Bind socket using parameters from ai.
  // If success, leave for loop, else try next entry.
}
```

Questions?