| Картина, която съдържа текст, вентилатор, устройство  Описанието е генерирано автоматично | **ВИСШЕ ВОЕННОМОРСКО УЧИЛИЩЕ „Н. Й. ВАПЦАРОВ“**  ***9002 Варна, ул. „В. Друмев“ 73, тел.052/632-015, факс 052/303-163*** |
| --- | --- |
| ***“FILII MARIS SUMUS”***  **ФАКУЛТЕТ „ИНЖЕНЕРЕН“**  **КАТЕДРА „ИНФОРМАЦИОННИ ТЕХНОЛОГИИ”** |
| A picture containing text  Description automatically generated |  |

**KУРСОВ ПРОЕКТ**

**по дисциплината „Алгоритмизация и основи на програмирането“**

**ТЕМА: „Direct Messaging System: Implementing a private messaging feature between users.“**



# Въведение

В този доклад ще разгледаме примерен проект на тази тема който има единствено за цел да се осъществи комуникация между клиент и сървър.Тази технология е в основата на множество модерни приложения, които улесняват ефективната и удобна комуникация между потребителите.

## Основни функционалности, които ще бъдат реализирани.

Системата за директни съобщения е проектирана да включва следните основни функции:

1. **Предаване на съобщения**: Клиентите да изпращат текстови съобщения до сървъра, който ги обработва
2. **Получаване на съобщения**: Позволяване както на клиентите, така и на сървъра да получават и показват входящи съобщения в реално време.
3. **Потребителска идентификация**: Прилагане основен механизъм за етикетиране, за разграничаване на съобщенията, изпратени от клиента и сървъра.
4. **Обработка на грешки**: Осигуряваме на механизми за откриване и обработка на неуспешни връзки или неочаквано въвеждане по изящен начин.

Като се фокусираме върху тези функционалности, проектът има за цел да създаде стабилен и функционален прототип за директни съобщения.

# Техническо описание

Системата е базирана на клиент-сървър архитектура, която осигурява ефективна и организирана комуникация между участниците. Архитектурата е структурирана по следния начин:

1. **Сървър**:
   * Слуша за входящи връзки от клиенти,
   * Приема клиентски връзки и управлява маршрутизирането на съобщения между свързани клиенти.
2. **Клиент**:
   * Изпраща въведени от потребителя съобщения до сървъра чрез **TCP** сокет.
   * Получава и показва съобщения, препратени от сървъра в реално време.

### Реализация на сървърът

| int main(void) {  int sockfd = socket(AF\_INET, SOCK\_STREAM, 0);  if (sockfd < 0) {  printf("Socket creation");   return 1;  }  struct sockaddr\_in address = {  AF\_INET,  htons(9999),  0  };  bind(sockfd, (struct sockaddr \*)&address, sizeof(address));   listen(sockfd, 10);  int clientfd = accept(sockfd, 0, 0);  *//stdin - 0 "input from the user"*  struct pollfd fds[2] = {  {  0,  POLLIN,  0  },  {  clientfd,  POLLIN,  0  }  };  for(;;) {  char buffer[256] = { 0 };  poll(fds, 2, 50000);   if (fds[0].revents & POLLIN) {  read(0, buffer, 255);  char label[300] = "Server: ";  strncat(label, buffer, sizeof(label) - strlen(label) - 1);  send(clientfd, label, strlen(label), 0);  }   else if (fds[1].revents & POLLIN) {  if (recv(clientfd, buffer, 255, 0) == 0) {  return 0;  }  printf("%s", buffer);  }  }  return 0; } |
| --- |

### Програмиране на сокет:

1. socket() - Осигурява основата за комуникация между клиент и сървър;
   * Аргументът **домейн** указва комуникационен домейн; това избира фамилията протоколи, която ще се използва за комуникация. Тези семейства са дефинирани в библиотеката <sys/socket.h>. Форматът, който се разбира от ядрото на Linux, и които ще използваме е: AF\_INET който представлява IPv4 или Internet Protocol.
   * Сокетът има аргументът **тип**, който определя типът на комуникационната . Дефинираният в момента тип е SOCK\_STREAM, който осигурява последователни, надеждни, двупосочни, базирани на връзка потоци от байтове.
   * Последният аргумент int protocol поставяме 0 което ни дава default протоколът за типът на домейна (TCP).
2. bind() - Позволява на сокета да се свърже към определен адрес, така че той действително да може да започне да го слуша;
   * Когато сокет е създаден, той съществува в пространство от имена (семейството от адреси), но няма присвоен адрес към него. bind() присвоява адреса, определен от const struct sockaddr addr, към сокета, посочен от файловия дескриптор int sockfd. addrlen указва размера в байтове на адресната структура, към която сочи const struct sockaddr addr. Традиционно тази операция се нарича „задаване на име на сокет“.
   * Дефинираме структура const struct sockaddr addr която отговаря за Internet domain sockets което е точно това от което имаме нужда;

| struct sockaddr\_in address = {  AF\_INET,  htons(9999), */\* номер на порта \*/*  0 */\* IPv4 адрес \*/* *//0 = localhost*  }; |
| --- |

1. listen() - Приготвя сокета за приемане на връзки;
   * Маркира сокета, посочен от sockfd, като пасивен сокет, т.е. като сокет, който ще се използва за приемане на входящи заявки за връзка.
   * Аргументът backlog дефинира максималната дължина, до която може да нарасне опашката от чакащи връзки за sockfd. Ако пристигне заявка за връзка, когато опашката е пълна, клиентът може да получи грешка или, ако основният протокол поддържа повторно предаване, заявката може да бъде игнорирана, така че по-късен повторен опит за свързване да успее.
2. accept() - извлича първата заявка за връзка от опашката от чакащи връзки за слушащия сокет
   * sockfd, създава нов свързан сокет и връща нов файлов дескриптор, отнасящ се до този сокет.
   * Последните два аргумента могат да върнат адрес, ако искаме адресът на клиента, но ние сме използваме localhost за това поставяме стойност 0 и на двете. *Ако изпълнението е успешно, accept() ще върне клиентския файлов дескриптор;*

## poll() System call

#include <poll.h> След като сървърът е приел връзката с клиента, трябва да се изпълнят 2 задачи:

1. Да се изпише всичко което клиентът изпраща на сървърът. Тоест ако клиентът изпраща чат съобщение, сървърът трябва да го изведе;
2. Да позволим на човекът, който управлява сървърът, да изпраща съобщения обратно към клиентът. Ще изпълним тези две задачи последователно. И начинът по който ще го направим това е да използваме функцията наречена poll();

poll()- Позволява на управляваните от събития I/O операции чрез наблюдение на множество файлови дескриптори за готовност.

Функцията poll() ни позволява да "изчакаме някакво събитие на даден файлов дескриптор". Също такта позволява на системата да обработва едновременно въвеждане от потребителя и входящи съобщения от мрежата.

| #include <poll.h> |
| --- |

| int poll(struct pollfd \*fds, nfds\_t nfds, int timeout); |
| --- |

*За да вземем вход от потребителя ще трябва да използваме stdin файлов дескриптор който винаги ще бъде равен на 0;*

1. Първият параметър ще бъде масив от pollfd структури:

| struct pollfd {  int fd; */\* file descriptor \*/* *//stdin = 0*  short events; */\* requested events \*/*  short revents; */\* returned events \*/* }; |
| --- |

* Ще ни трябват 2 от тези структури:
  1. Слушаме на stdin
  2. Слушаме на файловия дескриптор за да видим дали има съобщения идващи от клиента;

| struct pollfd fds[2] = {  {  0, *//stdin = 0*  POLLIN, *//There is data to be read*  0 *//It will be returned after the poll finishes*  },  {  clientfd, *//clientfd*  POLLIN,*//There is data to be read*  0 *//It will be returned after the poll finishes*  } }; |
| --- |

Сега можем да подадем структурата като първи аргумент на poll() 2. nfds\_t nfds - броят на файловите дескриптори 3. int timeout - време за изпълнение в милисекунди

#### Управление на входа и изхода

| for(;;) {  char buffer[256] = { 0 };  poll(fds, 2, 50000);    *//проверява дали revents съдържа POLLIN евента;*  *//и проверява се дали първият файлов дескриптор съдържа това*  if (fds[0].revents & POLLIN) {   read(0, buffer, 255);  char label[300] = "Server: ";  strncat(label, buffer, sizeof(label) - strlen(label) - 1);  send(clientfd, label, strlen(label), 0);  }   else if (fds[1].revents & POLLIN) {  if (recv(clientfd, buffer, 255, 0) == 0) {  return 0;  }  printf("%s", buffer);  }  } |
| --- |

Създаваме безкраен цикъл за да се поддържа постоянна комуникация в нея основно ще използваме тези елементи:

1. char buffer[256] = {0} - буферна променлива;
2. poll(fds, 2, 50000)- изчакаме някакво събитие на даден файлов дескриптор;
3. .revents - **return event** от poll() структурите;
   * Полето revents е изходен параметър, попълнен от ядрото на операционната система с действително настъпилите събития. Битовете, върнати в revents, могат да включват всеки от посочените в събития или една от стойностите POLLERR, POLLHUP, POLLNVAL, POLLIN;
4. read() - чете от stdin;

| ssize\_t read(int fd, void buf[.count], size\_t count); |
| --- |

- Опитва се да прочете до count байтове от файловия дескриптор fd в буфера, започвайки от buf;

1. send() - използва се за предаване на съобщение до друг сокет.

| ssize\_t send(int sockfd, const void buf[.len], size\_t len, int flags); |
| --- |

* sockfd - избрания сокет;
* buf[.len] - самото съобщение;
* len - размерът на съобщението;
* flags - нямаме нужда от флагове следователно поставяме 0;

*Ако предишният if не е верен тогава проверяваме следващият файлов дескриптор. И ако е верен, това означава, че има съобщение което е готово да се прочете, изпратено от клиентът;*

*Вложеният if проверява дали потребителя не получава нищо. Ако да да спре програмата;* 6. recv() - използва се за получаване на съобщения от сокет.

| ssize\_t recv(int sockfd, void buf[.len], size\_t len, int flags); |
| --- |

# Реализация на клиента

| int main(void) {  int sockfd = socket(AF\_INET, SOCK\_STREAM, 0);  if (sockfd < 0) {  printf("failed");   return 1;  }  struct sockaddr\_in address = {  AF\_INET,  htons(9999),  0  };  connect(sockfd, (struct sockaddr \*)&address, sizeof(address));  struct pollfd fds[2] = {  0,  POLLIN,  0  },  {  sockfd,  POLLIN,  0  }  };  for(;;) {  char buffer[256] = { 0 };  poll(fds, 2, 50000);   if (fds[0].revents & POLLIN) {  read(0, buffer, 255);  char labeled\_message[300] = "Client: ";  strncat(labeled\_message, buffer, sizeof(labeled\_message) - strlen(labeled\_message) - 1);  send(sockfd, labeled\_message, strlen(labeled\_message), 0);  }   else if (fds[1].revents & POLLIN) {  if (recv(sockfd, buffer, 255, 0) == 0) {  return 0;  }  printf("%s", buffer);  }  }  return 0; } |
| --- |

*Общо взето кодът остава същият.*

1. Вместо bind() ще използваме системното повикване connect() ;

| int connect(int sockfd, const struct sockaddr \*addr,   socklen\_t addrlen); |
| --- |

* Системното извикване connect() свързва сокета, посочен от файловия дескриптор sockfd, с адреса, указан от addr.
* Аргументът addrlen указва размера на addr. Форматът на адреса i addr се определя от адресното пространство на сокета sockfd; вижте socket(2) за повече подробности.

| *//необходими библиотеки* #include <stdio.h> //за printf(); #include <sys/socket.h> //за програмирането на сокета #include <string.h> //за работа със стрингове #include <arpa/inet.h> //за htons();   *//превръща shortint в nework byte order* #include <poll.h> //за poll(); #include <unistd.h> //за read(); |
| --- |

Компилираме сървърът и клиентът;

1. Отворете два терминални прозореца;
2. Първо стартирайте ./server;
3. След това ./client.

# Заключение

Като заключение мога да обобщя, че за мен този тип проект беше много интересен за изработване. Изисква базови знания за това как работят протоколи като **TCP**, как работят различни видове системни повиквания и работа със сокети.

Програмата е далеч от съвършена и малко безсмислена след като е свързана само на localhost. Може да се подобри като се използват реални IP адреси на реални компютърни системи от различни мрежи, но за това трябва да се преодолее **firewall** за който все още не знам как да се справя с него. Може също да се създаде графичен интерфейс за да бъде достъпен до повече хора.