Programação de Rede TCP/IP

[Aula 2. Início.]

Para esta aula (evidentemente) devem ler esta apostila. Além disso, recomendo que leiam o livro "Beej's Guide to Network Programming Using Internet Sockets", disponível em:
https://beej.us/guide/bgnet/ (acessado em 10/10/2024)

Resumo: Durante duas aulas, desenvolveremos programas que fazem comunicação entre Raspberry e computador. Faremos Raspberry enviar imagens da câmera para computador. Faremos controle manual dos motores do carrinho a partir do computador (fases 1 e 2 do projeto).

Nota: A partir desta aula, vocês vão desenvolver programas. Para não correr risco de perder os programas desenvolvidos, tirem backups. No mínimo, copiem todos os programas do computador no Raspberry e todos os programas do Raspberry no computador. Além disso, é recomendável copiar os programas num dispositivo externo (pendrive, HD externo ou na nuvem).

1 Comunicação TCP/IP

1.1 Introdução

Nas fases 1 e 2 do projeto, vamos fazer um programa que controla o carrinho remotamente, onde o usuário visualiza no computador os quadros capturados pela câmera da Raspberry e controla manualmente os motores do carrinho.

Para isso, vamos criar rotinas para transmitir vídeos com compactação quadro-a-quadro por JPG. Escolhendo uma taxa de compressão adequada, a compressão reduz em aproximadamente 10 vezes a quantidade de dados a ser transmitida. Além disso, este esquema permite que o processamento de visão computacional (fases 3-7 do projeto) seja feito no computador remoto, com maior poder de processamento.

Vamos usar o livro "Beej's Guide to Network Programming Using Internet Sockets", disponível gratuitamente em http://beej.us/guide/bgnet para entender transmissão de dados via internet. Leia o livro e procure entender os seguintes conceitos.

a) Socket.
b) Endereço IP versão 4 e 6.
c) Número da porta.
d) Ordem de byte (little-endian e big-endian).
e) UDP (User Datagram Protocol).
f) TCP (Transmisson Control Protocol).
g) Rede privada.

Além disso, procurem entender os seguintes conceitos usando outras fontes de informação:

h) Endereço MAC (media access control address): Veja, por exemplo:
https://en.wikipedia.org/wiki/MAC_address

i) NAT (Network Address Translation):
https://vnt-software.com/network-address-translation/

1.2 Velocidade de transmissão wifi

Vamos verificar qual é a velocidade de transmissão real entre o Raspberry e o computador. Podemos medir a velocidade de conexão entre os dois dispositivos usando o comando *iperf*:

```
$ sudo apt-get install iperf (no Raspberry e no computador)
pi@raspberrypi$ iperf -s (no servidor, isto é, Raspberry)
user@computer$ iperf -c 192.168.0.110 (no computador cliente)
```

onde 192.168.0.110 é o endereço IP do Raspberry.

Fazendo isto, obtive:

a) 13 Mbps, quando computador e Raspberry estão ligados através de um roteador antigo (de 2017, frequência 2.4GHz).
b) 35 Mbps, usando um roteador novo (de 2021) onde o computador está usando wifi 5GHz e Raspberry 2.4GHz. A mesma velocidade de 35 Mbps também é obtido usando VirtualBox dentro de Windows.
c) 163 Mbps, colocando um adaptador wifi USB no Raspberry e tanto o computador como o Raspberry estão ligados ao roteador pelo wifi 5GHz. Usei adaptador USB AC1300 (Archer T3U) da Tp-link.

Nota: As velocidades obtidas experimentalmente são muito menores que as velocidades comumente anunciadas pelos provedores de internet (300Mbps, 500Mbps, etc).
Nota: Fazendo as duas ligações (roteador-computador e roteador-raspberry) com cabos ethernet, obtive 94 Mbps (a porta ethernet do Raspberry v3 é de 100 Mbps).
Nota: Ligando dois computadores diretamente com cabo ethernet, obtive 936 Mbps (as portas ethernet dos computadores são de 1 Gbps) [https://askubuntu.com/questions/169473/sharing-connection-to-other-pcs-via-wired-ethernet].
Nota: Na sala GD-06, obtive velocidade de 31 Mbps tanto entre Raspberry e meu notebook (usando wifi) como entre Raspberry e computador da sala (usando conexão cabeada).

Nota: Se a taxa de transmissão estiver baixa, experimente rodar o aplicativo "wifi analyzer" em Android (figura 1). Pode ser que o canal de wifi do seu roteador esteja ocupado por outro dispositivo com sinal mais forte.

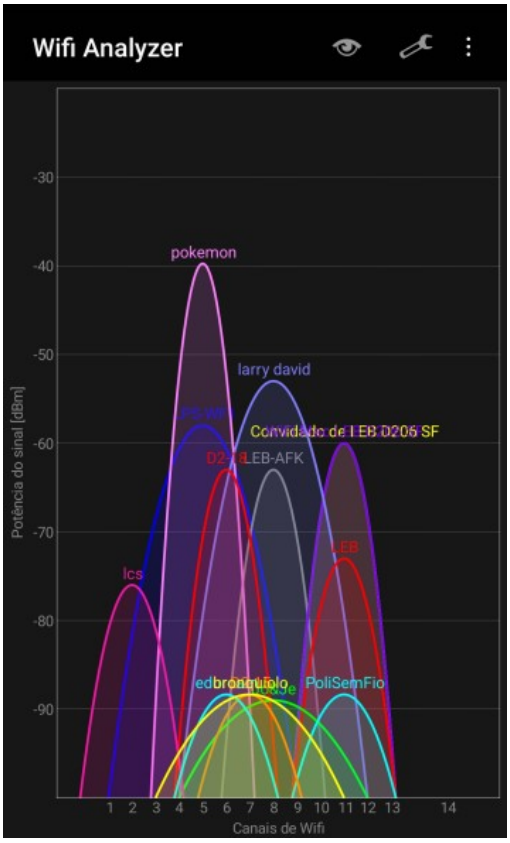


Figura 1: Tela de "wifi analyzer" de Android.

Uma imagem colorida 480×640 descompactada ocupa aproximadamente 1 MBytes ou 8 Mbits. Para transmitir 30 imagens coloridas 480×640 por segundo sem compressão, precisaria de uma comunicação de 240 Mbps. Em 35 Mbps passam apenas 4 quadros/s e em 13 Mbps passam apenas 1,6 quadros/s, o que tornaria o sistema pouco responsivo. Como transmitir uma quantidade maior de quadros? Uma solução é escrevermos um programa próprio para transmitir vídeos, fazendo compactação quadro-a-quadro do vídeo por JPEG (conhecida como "motion JPEG" ou MJPEG [wiki-MJPEG]).

Escolhendo uma taxa de compressão adequada, a quantidade de dados a ser transmitida diminui em aproximadamente 10 vezes. Além disso, este esquema permite que o processamento de visão computacional seja feito no computador remoto, que possui maior poder de processamento que Raspberry. Utilizando esta solução, consegui transmitir imagens coloridas 480×640 a 11 fps (roteador de 2017), 16 fps (roteador de 2021) e 20 fps (usando adaptador USB). No projeto, vamos usar imagens 240×320, o que permitirá aumentar ainda mais fps.

Nota: A fps não aumenta de forma inversamente proporcional à velocidade da conexão, pois há outros gargalos no processo, como o tempo para compactar imagem.

Nota: Usando compactação de vídeo (mpeg, divx, H.264, etc), é possível atingir taxas de compressão maiores. Porém, isso faz aumentar a latência e é pouco robusto a mudanças bruscas de conteúdo da imagem [http://www.spu.co.id/news/11/MJPEG-MPEG4-or-H-264-which-one-is-better].

Nota: Se algum aluno quiser, pode testar compactar vídeo – parece que [https://www.tutorialspoint.com/how-to-store-video-on-your-computer-in-opencv-using-cplusplus] mostra como gravar vídeo codificado na memória (para depois poder transmiti-lo).

Nota para mim: "sudo arp" mostra IP da conexão access point. Testar nmap e mtr.

1.3 Camadas da internet

O que geralmente chamamos de internet (rede de computadores) é composto por diversas camadas. As sete camadas tradicionais de uma rede são divididas em dois grupos: camadas superiores (camadas 5-7 da figura 2) e camadas inferiores (camadas 1-4). Vamos trabalhar no nível de soquete (em laranja). A interface de soquetes fornece uma API (Application Programming Interface) uniforme para acessar as camadas inferiores da rede. As camadas superiores não serão objetos do nosso estudo. [https://en.wikipedia.org/wiki/OSI_model]

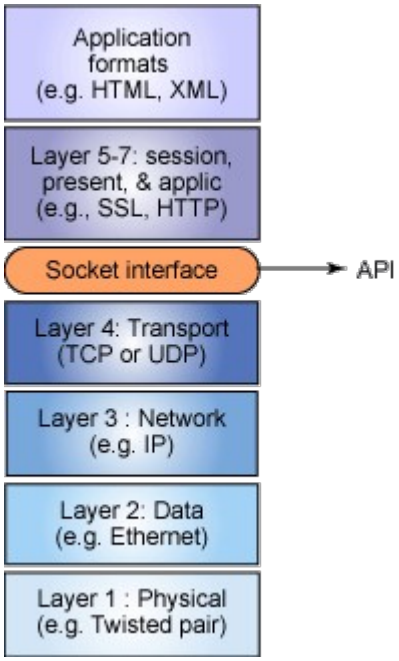


Figura 2: Camadas da rede [https://developer.ibm.com/tutorials/l-sock/]

A camada 1 (camada física) é a conexão física entre os nós, composta por conectores, cabos elétricos, fibras óticas, conexões sem fio, etc.

A camada 2 (camada de ligação de dados) permite transferir dados entre dois nós diretamente conectados.

A camada 3 (camada de rede) permite transferir pacotes de dados de um nó para outro nó, mesmo que não estejam diretamente conectados. Cada nó da rede tem um endereço IP (internet protocol) e pode transferir mensagens para outros nós apenas fornecendo o conteúdo da mensagem e o endereço IP do destino. A própria rede irá encontrar o caminho para entregar a mensagem ao nó destino, possivelmente roteando a mensagem através de nós intermediários.

A camada 4 (camada de transporte) é responsável por garantir a entrega confiável e correta dos dados entre os sistemas. Usa mecanismos como correção de erros e confirmação de recebimento para garantir que os dados cheguem ao destino corretamente.

As camadas 3 e 4 serão detalhadas nas próximas seções. Em vez de usarmos as camadas superiores prontas (camadas 5-7), construiremos a nossa própria aplicação.

1.4 UDP/IP e TCP/IP

1.4.1 IP - Internet Protocol

IP significa *internet protocol* (camada 3 da figura 2). Pacotes de dados na internet são enviados e recebidos entre dispositivos usando o protocolo IP. Cada pacote IP contém um cabeçalho e dados. O cabeçalho inclui endereço IP de origem e de destino, o tamanho do pacote e outros campos que ajudam rotear o pacote até o destino. Os dados são o conteúdo útil, como os caracteres de um email, parte de uma página web, ou a imagem compactada que queremos transmitir. Repare na figura 3 que o comprimento do pacote é descrito usando uma variável de 16 bits, o que faz com que o tamanho de um pacote IP não possa ser maior que 65.535 bytes.

Também repare que o endereço IP (v4) é uma variável de 32 bits, o que limita a “apenas” 4.294.967.296 o número de dispositivos que podem ser conectados na internet. Como o número de seres humanos na Terra é de aproximadamente 7 bilhões, não é possível fornecer um endereço IPv4 para cada habitante do planeta (muito menos se cada pessoa possui vários dispositivos conectados à internet). Procure descobrir como esta limitação é superada atualmente usando redes privadas. O endereço IP v6 é de 128 bits, o que permite obter $3{,}4\times10^{38}$ endereços, porém aparentemente ainda não é muito utilizado (2024).

Existem dois principais protocolos de transmissão de dados implementados “por cima” de IP: UDP (User Datagram Protocol) e TCP (Transmission Control Protocol) – figura 2, camada 4. As principais características dos dois são descritas a seguir [https://www.geeksforgeeks.org/differences-between-tcp-and-udp/].

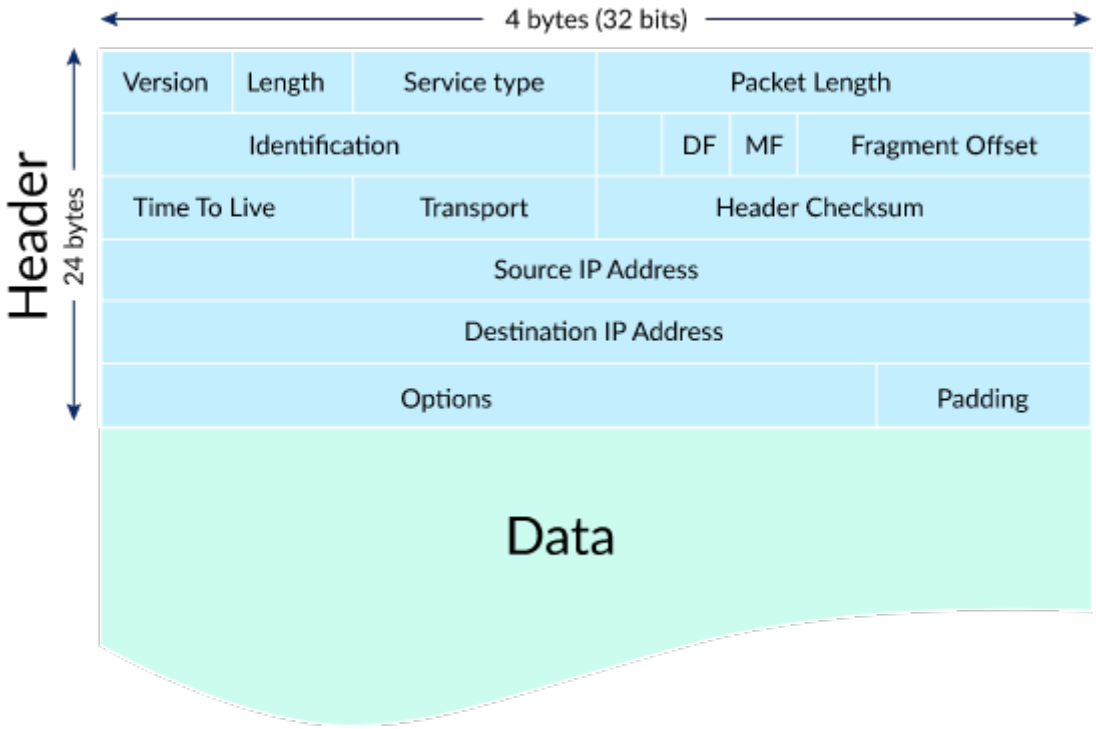


Figura 3: Pacote IP v4

https://www.khanacademy.org/computing/computers-and-internet/xcae6f4a7ff015e7d:the-internet/xcae6f4a7ff015e7d:routing-with-redundancy/a/ip-packets

1.4.2 UDP – User Datagram Protocol

No protocolo UDP, *talker* simplesmente envia mensagens para *listener*, especificando somente o endereço IP do *listener*, sem estabelecer uma conexão entre ambos. Se a mensagem chegar num instante em que *listener* não estava "prestando atenção", a mensagem se perde. As mensagens que *talker* envia podem chegar ao *listener* fora de ordem. UDP possui somente um mecanismo básico de checagem de erro, baseado em *checksums*.

Nota: Checksum ou soma de verificação é um código usado para verificar a integridade de dados transmitidos através de um canal com ruídos ou armazenados em algum meio [WikiSoma].

É possível que, numa rede lenta, *talker* fique enviando uma mensagem após outra, entupindo a rede. Por outro lado, UDP é mais rápido e eficiente que TCP. É adequado, por exemplo, para enviar vídeo em tempo real (como na nossa aplicação) onde interessa que os quadros cheguem sem atraso e não tem muita importância perder alguns quadros. Veja na figura 4 que UDP acrescenta um cabeçalho próprio antes de inserir os dados no pacote IP. Assim, a quantidade máxima de bytes úteis num pacote UDP seria 65.535-*h*, onde *h* é o tamanho dos cabeçalho de UDP/IP.

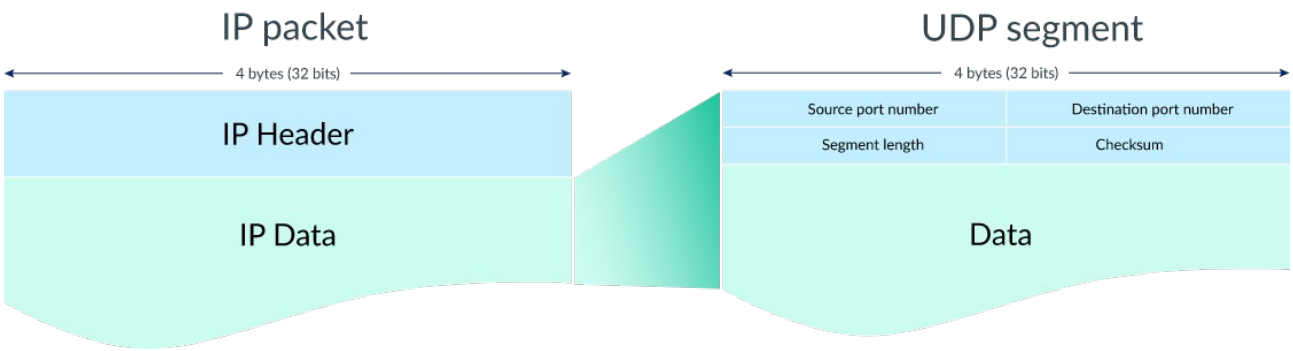


Figura 4: Pacote UDP [Khanacademy]

1.4.3 TCP – Transmission Control Protocol

Em TCP, é necessário estabelecer uma conexão entre os computadores emissor e receptor antes de iniciar a comunicação, e é necessário fechar a conexão no fim. TCP é confiável, pois garante que todos os dados chegarão ao destino na ordem correta. Possui vários mecanismos de checagem de erro. Se um pacote chegar ao destino com erro, o protocolo requisita automaticamente que o emissor envie novamente o pacote de forma transparente para o usuário final. Estabelecendo uma troca de mensagens simples entre os dois computadores, os computadores passam automaticamente a enviar menos dados quando a rede fica congestionada [StackUDP]. A figura 5 mostra o cabeçalho TCP – repare como possui mais informações que o cabeçalho UDP.

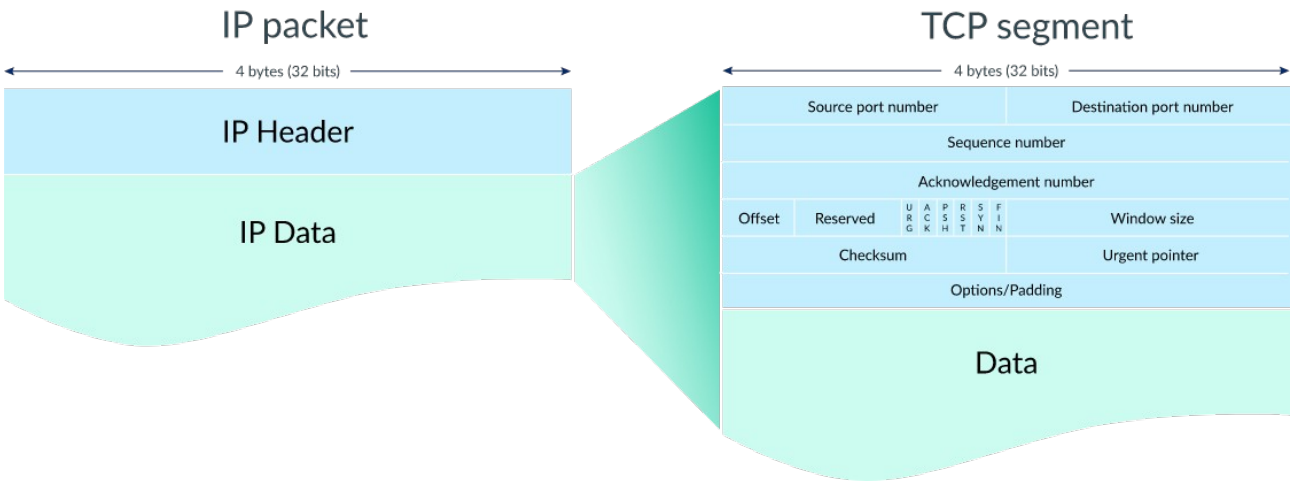


Figura 5: Pacote TCP [KhanAcademy]

Pelas características da nossa aplicação, o mais adequado seria usar UDP, pois interessa-nos alta velocidade de transmissão, baixa latência e não tem importância perder alguns quadros. O problema é que, como UDP não é confiável, o programador (isto é, você) deve escrever vários mecanismos para torná-lo confiável, tornando o projeto complexo. A chance destes mecanismos não funcionarem é bastante alta. Como todo o projeto depende da comunicação entre Raspberry e computador, poderia acontecer que muitos alunos não consigam avançar além deste ponto no projeto. Assim, vamos usar o protocolo TCP no projeto, mesmo sabendo que provavelmente UDP seria a escolha mais adequada.

1.4.4 Número de porta

Tanto UDP quanto TCP acrescentam os números de porta de origem e de destino nos cabeçalhos. O número de porta identifica uma aplicação ou um serviço do sistema. Por exemplo, porta 22 é usada por SSH, 25 por SMTP (email), 80 por HTTP, 443 por HTTPS, etc. Assim, só os pacotes de SSH (por exemplo) são entregues ao aplicativo de SSH. As portas 49152-65535 estão livre para serem usadas em aplicações dos usuários segundo [http://www.steves-internet-guide.com/tcpip-ports-sockets] ou todas as portas acima de 1024 estão livres segundo [https://beej.us/guide/bgnet]. No nosso projeto, vamos usar a porta 3490, seguindo o exemplo do livro.

1.5 Sistema que transmite/recebe mensagens usando TCP/IP

Para transmitir/receber dados via TCP/IP, vamos usar o livro "Beej's Guide to Network Programming Using Internet Sockets", disponível gratuitamente em:
`http://beej.us/guide/bgnet`

1) Leia o livro para entender como funcionam UDP/IP e TCP/IP, as estruturas de dados envolvidos, e as funções disponíveis em Linux.

2) No capítulo 6, há um exemplo de conexão UDP/IP chamado talker-listener. Baixe os programas, compile-os e execute-os (por exemplo, configurando Raspberry como talker e Computador como listener).

3) No capítulo 6, há um exemplo de conexão TCP/IP server-client. Baixe os programas *server.c* e *client.c*, compile-os e execute-os.

Iremos modificar *server.c* e *client.c* do livro para transmitir o vídeo capturado pela Raspberry ao computador e os comandos do computador ao Raspberry. Para facilitar esse trabalho, listo abaixo *server5.c* e *client5.c*, onde fiz duas modificações:

a) O programa original *server.c* estava preparado para servir ao mesmo tempo vários clientes. Isto gera várias complicações desnecessárias ao nosso projeto, pois no nosso projeto a comunicação é somente entre Raspberry e Computador. Modifiquei-o para servir a um único cliente.
b) Modifiquei os programas *server.c* e *client.c* para tanto enviar como receber dados, para deixar o exemplo mais completo.

Exercício altamente recomendado: Compile e execute server5.c/client5.c:[1]

```
Raspberry> server5
Computador> client5 192.168.0.110
```

Raspberry deve responder:

```
server: waiting for connections...
server: got connection from 192.168.0.109
server: received 'Mensagem #2 do client para server'
```

E o computador deve responder:

```
client: connecting to 192.168.68.110
client: received 'Mensagem #1 do server para client'
client: received 'Mensagem #3 do server para client'
```

1 É possível fazer teste em duas janelas de um computador (sem usar Raspberry) usando IP "loopback" 127.0.0.1:

```
Computador-terminal1$ server5
Computador-terminal2$ client5 127.0.0.1
```

```c
/*
** server5.c -- a stream socket server demo
** Modificado pelo Hae para atender um unico cliente.
** compila server5
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <arpa/inet.h>
#include <sys/wait.h>
#include <signal.h>

#define PORT "3490"  // the port users will be connecting to
#define BACKLOG 1   // how many pending connections queue will hold
#define MAXDATASIZE 256 // max number of bytes we can get at once

// get sockaddr, IPv4 or IPv6:
void *get_in_addr(struct sockaddr *sa) {
  if (sa->sa_family == AF_INET) {
    return &(((struct sockaddr_in*)sa)->sin_addr);
  }
  return &(((struct sockaddr_in6*)sa)->sin6_addr);
}

int main(void) {
  int sockfd, new_fd;  // listen on sock_fd, new connection on new_fd
  char buf[MAXDATASIZE];
  struct addrinfo hints, *servinfo, *p;
  struct sockaddr_storage their_addr; // connector's address information
  socklen_t sin_size;
  struct sigaction sa;
  int yes=1;
  char s[INET6_ADDRSTRLEN];
  int rv;

  memset(&hints, 0, sizeof hints);
  hints.ai_family = AF_UNSPEC;
  hints.ai_socktype = SOCK_STREAM;
  hints.ai_flags = AI_PASSIVE; // use my IP

  if ((rv = getaddrinfo(NULL, PORT, &hints, &servinfo)) != 0) {
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv));
    return 1;
  }

  // loop through all the results and bind to the first we can
  for(p = servinfo; p != NULL; p = p->ai_next) {
    if ((sockfd = socket(p->ai_family, p->ai_socktype,
        p->ai_protocol)) == -1) {
      perror("server: socket");
      continue;
    }
    if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &yes,
        sizeof(int)) == -1) {
      perror("setsockopt");
      exit(1);
    }
    if (bind(sockfd, p->ai_addr, p->ai_addrlen) == -1) {
      close(sockfd);
      perror("server: bind");
      continue;
    }
    break;
  }
  freeaddrinfo(servinfo); // all done with this structure
  if (p == NULL)  {
    fprintf(stderr, "server: failed to bind\n");
    exit(1);
  }

  if (listen(sockfd, BACKLOG) == -1) {
    perror("listen");
    exit(1);
  }
  printf("server: waiting for connections...\n");

  while (1) {
    sin_size = sizeof their_addr;
    new_fd = accept(sockfd, (struct sockaddr *)&their_addr, &sin_size);
```

```
    if (new_fd == -1) {
      perror("accept");
      continue;
    } else break;
  }

  inet_ntop(their_addr.ss_family,
    get_in_addr((struct sockaddr *)&their_addr),
    s, sizeof s);
  printf("server: got connection from %s\n", s);
  close(sockfd); // doesn't need the listener anymore

  strcpy(buf,"Mensagem #1 do server para client");
  if (send(new_fd, buf, strlen(buf)+1, 0) == -1) perror("send");

  if (recv(new_fd, buf, MAXDATASIZE, 0) == -1) perror("recv");
  printf("server: received '%s'\n",buf);

  strcpy(buf,"Mensagem #3 do server para client");
  if (send(new_fd, buf, strlen(buf)+1, 0) == -1) perror("send");

  close(new_fd);
  return 0;
}
```

```c
/*
** client5.c -- a stream socket client demo
** Modificado pelo Hae para receber e transmitir dados
** compila client5
*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <arpa/inet.h>

#define PORT "3490" // the port client will be connecting to
#define MAXDATASIZE 256 // max number of bytes we can get at once

// get sockaddr, IPv4 or IPv6:
void *get_in_addr(struct sockaddr *sa) {
  if (sa->sa_family == AF_INET) {
    return &(((struct sockaddr_in*)sa)->sin_addr);
  }
  return &(((struct sockaddr_in6*)sa)->sin6_addr);
}

int main(int argc, char *argv[]) {
  int sockfd, numbytes;
  char buf[MAXDATASIZE];
  struct addrinfo hints, *servinfo, *p;
  int rv;
  char s[INET6_ADDRSTRLEN];

  if (argc != 2) {
    fprintf(stderr,"usage: client hostname\n");
    exit(1);
  }

  memset(&hints, 0, sizeof hints);
  hints.ai_family = AF_UNSPEC;
  hints.ai_socktype = SOCK_STREAM;
  if ((rv = getaddrinfo(argv[1], PORT, &hints, &servinfo)) != 0) {
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv));
    return 1;
  }

  // loop through all the results and connect to the first we can
  for(p = servinfo; p != NULL; p = p->ai_next) {
    if ((sockfd = socket(p->ai_family, p->ai_socktype,
        p->ai_protocol)) == -1) {
      perror("client: socket");
      continue;
    }
    if (connect(sockfd, p->ai_addr, p->ai_addrlen) == -1) {
      perror("client: connect");
      close(sockfd);
      continue;
    }
    break;
  }

  if (p == NULL) {
    fprintf(stderr, "client: failed to connect\n");
    return 2;
  }

  inet_ntop(p->ai_family, get_in_addr((struct sockaddr *)p->ai_addr),
      s, sizeof s);
  printf("client: connecting to %s\n", s);
  freeaddrinfo(servinfo); // all done with this structure

  if (recv(sockfd, buf, MAXDATASIZE, 0) == -1) perror("recv");
  printf("client: received '%s'\n",buf);

  strcpy(buf,"Mensagem #2 do client para server");
  if (send(sockfd,buf,strlen(buf)+1,0)==-1) perror("send");

  if (recv(sockfd, buf, MAXDATASIZE, 0) == -1) perror("recv");
  printf("client: received '%s'\n",buf);

  close(sockfd);
  return 0;
}
```

2 Classes SERVER e CLIENT

Os programas *server5.c* e *client5.c* acima têm dois problemas:

1) O primeiro é que as funções *send()* e *recv()* do Linux possuem limitações. A função *send()* retorna o número de bytes realmente enviado - que pode ser menor do que o número de bytes que você pediu para enviar. Se a função *send()* tiver enviado menos bytes do que o pedido, você (isto é, o seu programa) deve se encarregar de enviar os bytes que ainda restam. O mesmo acontece com a função *recv* - pode não receber todos os bytes porque *send* enviou dados parciais. Neste caso, esta função deve ser chamada novamente para receber os bytes que restam. Essas funções retornam -1 em caso de erro. Você pode consultar o manual dessas funções escrevendo no terminal do Linux:

```
$ man send
$ man recv
```

2) O segundo é que os programas *server5.c* e *client5.c* (escritos na linguagem *C*) estão "uma bagunça". O autor do livro escreveu os programas de forma limpa, tomando bastante cuidado. Apesar disso, não é fácil utilizá-los dentro de outros programas, pois a linguagem *C* não deixa claro quais trechos dos programas devem ser copiados para dentro de novos programas.

No desenvolvimento de um sistema de software, quanto maior for a "bagunça", maior a chance de ter erros no sistema. Para "arrumar a bagunça", vamos converter os programas de *C* para *C++* e encapsular o código dentro de duas classes: SERVER e CLIENT. Depois de assegurar que estas classes funcionam sem erros, podemos deixá-los num arquivo "include" ou numa biblioteca pré-compilada e não mais nos preocuparmos com o seu funcionamento interno. Vamos escrever as classes SERVER e CLIENT em etapas.

Arquivo *raspberry.hpp*:

Este ano (2024), usaremos o pacote Cekeikon apenas para ter acesso à biblioteca OpenCV (no computador) e para chamar o programa *compila* (tanto no computador e como no Raspberry). Não linkaremos o projeto com a biblioteca Cekeikon. Porém, dentro da biblioteca Cekeikon, há funções e classes que podem ser úteis ao nosso projeto. Vamos copiar estas funções/classes num arquivo *raspberry.hpp* a ser incluído no projeto. Isto permitirá que o aluno tenha maior conhecimento do que está acontecendo dentro dessas funções/classes.

Neste momento, vamos colocar apenas quatro funções e uma definição de tipo em *raspberry.hpp*. Iremos colocando outras funções à medida em que avançarmos no projeto.

```
//raspberry.hpp
#include <cstdio>
#include <cstdlib>
#include <iostream>
#include <sstream>
#include <chrono>
using namespace std;

void erro(string s1="") {
  cerr << s1 << endl;
  exit(1);
}

#define xdebug { string st = "File="+string(__FILE__)+" line="+to_string(__LINE__)+"\n"; cout << st; }

#define xprint(x) { ostringstream os; os << #x " = " << x << '\n'; cout << os.str(); }

double timeSinceEpoch() {
  //calculates the time elapsed since the epoch (typically January 1, 1970, for most systems)
  //in seconds as a floating-point number of type double.
  using namespace std::chrono;
  return duration_cast<duration<double>>( system_clock::now().time_since_epoch() ).count();
}

typedef uint8_t BYTE;
```

1) A função
erro("Mensagem de erro")
imprime a mensagem de erro e aborta o programa. Equivale à função do Python:
sys.exit("Mensagem de erro")
No nosso projeto, se acontecer algum erro de execução, simplesmente vamos imprimir uma mensagem amigável e abortar o programa.

2) O macro
xdebug
imprime o nome do arquivo-fonte e a linha do código onde está o macro. Se o seu programa travar, colocar *xdebug* em várias partes do código permite descobrir até onde o programa chegou antes de travar.

3) O macro
xprint(variavel)
que imprime o nome do variável e o seu conteúdo.

Exemplo:

```
1  //teste.cpp
2  #include "raspberry.hpp"
3  int main() {
4    xdebug;
5    int a=1;
6    xprint(a);
7    xdebug;
8    string st="abc";
9    xprint(st);
10   erro("Erro: Vamos abortar programa aqui");
11   xdebug;
12 }
```

```
Saída:


File=teste.cpp line=4

a = 1
File=teste.cpp line=7

st = abc
Erro: Vamos abortar programa aqui
```

Para compilar e executar:

```
Computador/Raspberry$ compila teste
Computador/Raspberry$ ./teste
```

Nota: A diferença entre os comandos:

#include <arquivo.h>

#include "arquivo.h"

é que o primeiro comando procura *arquivo.h* nos diretórios definidos pelo compilador para os arquivos do tipo include. O segundo comando procura *arquivo.h* no seu diretório atual. No segundo caso, é possível especificar o diretório onde se encontra o *arquivo.h*, por exemplo:

#include "/home/usuario/labsist/proj2024/arquivo.h"

4) A função

timeSinceEpoch

retorna uma variável ponto flutuante precisão *double* com a quantidade de segundos que se passaram desde *epoch*. Pode ser usada para cronometrar o tempo de execução de um programa.

Nota: Em sistemas Linux, *epoch* equivale a 1 de janeiro de 1970, 00:00:00 UTC. Em Windows, *epoch* é 1 de janeiro de 1601.

Exemplo:

```
//timeSinceEpoch.cpp
#include "raspberry.hpp"
int main() {
  double t1=timeSinceEpoch();

  //Algum processamento para medir o tempo gasto
  double d=1;
  for (int i=0; i<100000000; i++) {
    d=d+i; d=d*0.0001;
  }
  cout << "Resposta: " << d << endl;

  double t2=timeSinceEpoch();
  cout << "Gastou " << t2-t1 << " segundos." << endl;
}
```

```
Saída:

Resposta: 10001

Gastou 0.183965 segundos.
```

5) Vamos definir o tipo BYTE como sinônimo de inteiro sem sinal de 8 bits.

```
typedef uint8_t BYTE;
```

2.1 Métodos *sendBytes* e *receiveBytes*

Escreveremos em primeiro lugar os métodos *sendBytes* e *receiveBytes* para mandar/receber *n* bytes, porque fica simples mandar/receber qualquer tipo de variável chamando internamente esses dois métodos.

Nota: Se a variável for simples (isto é, não for composta por cabeçalho e dado), basta enviar os bytes contidos dentro de um certo espaço de memória. Se a variável consistir de cabeçalho e dados, deve-se primeiro enviar as informações que permitam reconstruir o cabeçalho do outro lado e em seguida enviar os dados.

Escreva as classes SERVER e CLIENT com os métodos *sendBytes* e *receiveBytes* que permitem enviar/receber um número arbitrário de bytes. Os *headers* devem ser algo como:

```
class SERVER {
  ...
public:
  SERVER();
  ~SERVER();
  void waitConnection();
  void sendBytes(int nBytesToSend, BYTE *buf);
  void receiveBytes(int nBytesToReceive, BYTE *buf);
  ...
};

class CLIENT {
  ...
public:
  CLIENT(string endereco);
  ~CLIENT();
  void sendBytes(int nBytesToSend, BYTE *buf);
  void receiveBytes(int nBytesToReceive, BYTE *buf);
  ...
};
```

Você deve pegar os programas *server5.c* e *client5.c* e copiar os códigos que devem ser executados no início do programa dentro dos construtores *SERVER()* e *CLIENT()*. Pode escrever os códigos dentro ou fora dos headers, isto é:

```
class SERVER {
  ...
  SERVER() {
    //escreva o codigo aqui
  }
  ...
}
```

OU

```
class SERVER {
  ...
  SERVER();
  ...
};

SERVER::SERVER() {
  //escreva o codigo aqui
}
```

Você deve copiar o código a ser executado enquanto servidor espera que cliente solicite conexão dentro do método *waitConnection()*. Depois, deve colocar os códigos que fecham a conexão dentro dos destrutores *~SERVER()* e *~CLIENT()*.

Por fim, deve escrever os métodos *sendBytes* e *receiveBytes*.

Cuidado: Escreva essas classes e métodos com *muito* cuidado, assegurando-se de que não haja *nenhum* erro, pois os erros na programação de baixo nível são difíceis de detectar. Se estas funções tiverem qualquer errinho, o seu programa pode parecer que funciona bem durante muito tempo. Porém, pode dar erro em lugares e instantes completamente inesperados. É o pior tipo de erro, pois é muito difícil localizar a sua origem.

Observação: Não há necessidade de escrever cada classe num arquivo diferente.

Em sistemas de software, não é boa prática copiar o mesmo código em vários lugares diferentes do sistema. Neste caso, se tiver que fazer alguma alteração, vai ter que alterar todas as cópias dela, propiciando a introdução de erros. Como escreveremos vários programas que usam as classes SERVER e CLIENT, é recomendável deixar o código num único lugar. Para isso, a solução mais simples é guardar essas classes num arquivo “include”. Vamos chamar esse arquivo de *projeto.hpp* e incluir esse arquivo em todos os programas que vão utilizar essas classes com o comando.

Agora, temos dois arquivos include:

`1) "raspberry.hpp"`
contém funções/classes úteis. Este arquivo é público e todos os alunos têm acesso a ele.

`2) "projeto.hpp"`
contém funções/classes do seu projeto. Só o seu grupo têm acesso a este arquivo, pois uma boa parte do seu projeto estará nele. O primeiro comando deste arquivo deve ser incluir o arquivo "raspberry.hpp".

Além disso, coloque em *projeto.hpp* a função *testaBytes* abaixo. Esta função testa se todos os *n* bytes da memória apontada por *buf* possuem o valor *b*. Com isso, *projeto.hpp* fica:

```
//projeto.hpp - incluir nos programas do seu projeto
#include "raspberry.hpp"
(... outros includes ...)

(... outras classes como SERVER e CLIENT e funções ...)

bool testaBytes(BYTE* buf, BYTE b, int n) {
  //Testa se n bytes da memoria buf possuem valor b
  bool igual=true;
  for (unsigned i=0; i<n; i++)
    if (buf[i]!=b) { igual=false; break; }
  return igual;
}
```

No seu programa, só precisa incluir o arquivo projeto.hpp (não precisa incluir raspberry.hpp), pois projeto.hpp irá incluir raspberry.hpp.

Escrevendo corretamente as classes SERVER e CLIENT e a função *testaBytes* no arquivo *projeto.hpp*, os programas *server6b.cpp* e *client6b.cpp* abaixo devem funcionar:

```
//server6b.cpp – rodar no Raspberry
//testa sendBytes e receiveBytes
#include "projeto.hpp"

int main(void) {
  SERVER server;
  server.waitConnection();

  const int n=100000;
  BYTE buf[n];
  memset(buf,111,n); //insere 111 em n bytes a partir do endereço buf
  server.sendBytes(n,buf);

  server.receiveBytes(n,buf);
  if (testaBytes(buf,214,n)) printf("Recebeu corretamente %d bytes %d\n",n,214);
  else printf("Erro na recepcao de %d bytes %d\n",n,214);

  memset(buf,111,n);
  server.sendBytes(n,buf);
}
```

```
//client6b.cpp – rodar no computador
//testa sendBytes e receiveBytes
#include "projeto.hpp"

int main(int argc, char *argv[]) {
  if (argc!=2) erro("client6 servidorIpAddr\n");
  CLIENT client(argv[1]);

  const int n=100000;
  BYTE buf[n];

  client.receiveBytes(n,buf);
  if (testaBytes(buf,111,n)) printf("Recebeu corretamente %d bytes %d\n",n,111);
  else printf("Erro na recepcao de %d bytes %d\n",n,111);

  memset(buf,214,n); //insere 214 em n bytes a partir do endereço buf
  client.sendBytes(n,buf);

  client.receiveBytes(n,buf);
  if (testaBytes(buf,2,n)) printf("Recebeu corretamente %d bytes %d\n",n,2);
  else printf("Erro na recepcao de %d bytes %d\n",n,2);
}
```

[Lição de casa #2/2 da aula 2. Vale 5.] Escreva as classes SERVER e CLIENT com métodos *sendBytes* e *receiveBytes* e inclua função *testaBytes* dentro do arquivo *projeto.hpp* de modo que os programas *server6b.cpp* e *client6b.cpp* funcionem corretamente. Mostre *server6b/client6b* funcionando.

Para compilar os dois programas:

```
pi@raspberrypi$ compila server6b
usuario@computador$ compila client6b
```

Saída esperada:

```
pi@raspberrypi$ server6b
server: got connection from 192.168.0.109
Recebeu corretamente 100000 bytes 214

usuario@computador$ client6b 192.168.0.110
client: connecting to 192.168.0.110
Recebeu corretamente 100000 bytes 111
Erro na recepcao de 100000 bytes 2
```

O “erro” impresso na última linha é proposital.

Nota: Também é possível testar os programas abrindo dois terminais no computador, sem usar Raspberry. Para isso, use endereço IP de *loopback* 127.0.0.1.

```
terminal1$ server6b
server: Esperando conexao...
server: recebi conexao de 127.0.0.1
Recebeu corretamente 100000 bytes 214
server: fechei conexao.

terminal2$ client6b 127.0.0.1
client: connecting to 127.0.0.1
Recebeu corretamente 100000 bytes 111
Erro na recepcao de 100000 bytes 2
```

Ajuda da lição de casa #2: Classe *SERVER* fica algo como (você deve preencher os tres pontinhos):

```
//projeto.hpp
#include "raspberry.hpp"
#include <unistd.h>
...

class SERVER {
  const string PORT="3490";  // the port users will be connecting to
  const int BACKLOG=1;   // how many pending connections queue will hold
  int sockfd, new_fd;  // listen on sock_fd, new connection on new_fd
  struct addrinfo hints, *servinfo, *p;
  ...

  static void *get_in_addr(struct sockaddr *sa) {
    //Esta funcao esta repetida em SERVER e CLIENT.
    //E' possivel unificar usando DEVICE.
    ...
  }

public:
  SERVER() {
    memset(&hints, 0, sizeof hints);
    ...
  }

  ~SERVER() {
    close( ...
  }

  void waitConnection() {
    printf("server: waiting for connections...\n");
    while (1) {
    ...
  }

  void sendBytes(int nBytesToSend, BYTE *buf) {
    ...
  }

  void receiveBytes(int nBytesToReceive, BYTE *buf) {
    ...
  }
}; //Fim de SERVER
```

Continuação da ajuda da lição de casa #2: Classe *CLIENT* fica algo como (você deve preencher os três pontinhos, continuando o arquivo *projeto.hpp*):

```
//projeto.hpp (continuacao)
class CLIENT: public DEVICE {
  const string PORT="3490"; // the port client will be connecting to
  int sockfd, numbytes;
  ...

  static void *get_in_addr(struct sockaddr *sa) {
    //Esta funcao esta repetida em SERVER e CLIENT.
    //E' possivel unificar usando DEVICE.
    ...
  }
public:
  CLIENT(string endereco) {
    memset(&hints, 0, sizeof hints);
    ...
  }

  ~CLIENT() {
    close(sockfd);
  }

  void sendBytes(int nBytesToSend, BYTE *buf) {
    ...
  }

  void receiveBytes(int nBytesToReceive, BYTE *buf) {
    ...
  }
}; //Fim de CLIENT
```

Nota: Pedindo para imprimir o número de bytes enviados/recebidos pelas funções send/recv ao enviar 100000 bytes de Raspberry para Computador, obtive a seguinte saída.

```
pi@raspberrypi:~/labsist $ server6b
server: Esperando conexao...
server: recebi conexao de 192.168.68.109
SERVER nSentBytes=100000

(CekCPU)hae@Voyager:~/labsist$ client6b 192.168.68.110
client: connecting to 192.168.68.110
CLIENT nReceivedBytes=1448
CLIENT nReceivedBytes=5792
CLIENT nReceivedBytes=7240
CLIENT nReceivedBytes=5792
CLIENT nReceivedBytes=4344
CLIENT nReceivedBytes=4344
CLIENT nReceivedBytes=1448
CLIENT nReceivedBytes=10136
CLIENT nReceivedBytes=31856
CLIENT nReceivedBytes=2896
CLIENT nReceivedBytes=1448
CLIENT nReceivedBytes=4344
CLIENT nReceivedBytes=1448
CLIENT nReceivedBytes=1448
CLIENT nReceivedBytes=10136
CLIENT nReceivedBytes=4344
CLIENT nReceivedBytes=1536
Recebeu corretamente 100000 bytes 111
```

A função *send* conseguiu enviar 100000 bytes numa única chamada, enquanto que a função *recv* recebeu 100000 bytes em várias chamadas.

[Aula 2. Fim.]

[Aula 3. Início.] [Durante a aula, explicar somente as figuras desta seção.]

3 Programação de “baixo nível”

Nos anos anteriores, notei que muitos alunos enfrentam grandes dificuldades para concluir as tarefas desta aula. Percebi que não possuem conhecimento de programação de “baixo nível” em C/C++, mexendo nos “bits e bytes”. Escrevo esta seção na esperança de ajudá-los. Se algum aluno achar que domina este assunto, pode pular esta seção.

1) Um byte são 8 bits. Um byte pode representar um inteiro sem sinal de 0 a 255, um inteiro com sinal entre -128 e +127, um caracter, oito variáveis booleanas, etc. Quem deve interpretar o que os 8 bits representam é o próprio programador.

2) As variáveis de 8 bits podem ser com ou sem sinal.

As variáveis 8 bits com sinal são chamadas de char (C/C++) ou int8_t (C++).

As variáveis de 8 bits sem sinal em C/C++ podem receber diferentes nomes nas diferentes bibliotecas: unsigned char (C/C++), uchar (OpenCV), BYTE (Microsoft e Cekeikon), uint8_t (C++), GRY (Cekeikon), etc.

Daqui em diante, vamos chamar variáveis de 8 bits sem sinal de BYTE, pois este nome deixa bem claro que representa 8 bits sem sinal. Este tipo já está declarado no Cekeikon.

3) Pode-se imaginar toda a memória do computador como um longo *array* de bytes. Nesse *array* de bytes estão armazenados programas e dados.

4) Existem diferenças entre variáveis de C e de C++. Vamos estudar as diferenças entre *array* de C e *vector* de C++. Também vamos ver as diferenças entre *string* de C e *string* de C++.

5) Vamos recordar como funciona “array” de *C* e a sua relação com “pointer”. Considere o programa abaixo.

```
 1 //array.cpp
 2 #include <iostream>
 3 using namespace std;
 4
 5 int main() {
 6   int a[2]; a[0]=2; a[1]=4;
 7   cout << a[0] << endl;
 8   cout << a << endl;
 9
10   int *p; p=a;
11   cout << *p << endl;
12   cout << p[0] << endl;
13
14   cout << *(p+1) << endl;
15   cout << p[1] << endl;
16
17   uint8_t *b; b=(uint8_t*)(a);  // b=a nao funciona
18   cout << int(b[0]) << " " << int(b[1]) << " " << int(b[2]) << " " << int(b[3]) << endl;
19   cout << int(b[4]) << " " << int(b[5]) << " " << int(b[6]) << " " << int(b[7]) << endl;
20 }
```

Nome da variável	Saída
a[0]	2
a	0x7ffc63597060
*p	2
p[0]	2
*(p+1)	4
p[1]	4
b[0] b[1] b[2] b[3]	2 0 0 0
b[4] b[5] b[6] b[7]	4 0 0 0

Na linha 6, o programa cria um array *a* com 2 inteiros e coloca os números “2” e “4” nas posições “0” e “1”. Assim, se imprimir a[0] (linha 7), o número 2 é impresso. Porém, se pedir para imprimir *a* (linha 8) o endereço do primeiro elemento do array *a* será impresso.

Na linha 10, *p* é um apontador para inteiro e fazemos com que ele aponte para *a* atribuindo a ele o endereço de *a*. Depois, pedindo para imprimir **p* (apontado por *p*, linha 11) ou *p[0]* (o primeiro elemento do array de inteiro apontado por *p*, linha 12) será impresso número “2”. Se pedir para imprimir **(p+1)* ou *p[1]* (linhas 14 e 15) será impresso o segundo elemento do array ou “4”.

Cada inteiro ocupa 4 bytes e é possível acessar o array byte a byte, fazendo com que um apontador para byte *b* aponte para array *a*. A linha 17 faz isso, através de um *typecast* que converte o endereço de array de inteiro *a* para apontador de byte (uint8_t*).

Array *a* pode ser representado como tendo 2 inteiros (linha superior da figura 6), ou como tendo 8 bytes (linha inferior). Pedindo para imprimir os 8 bytes, obtemos os valores 2 0 0 0 4 0 0 0.

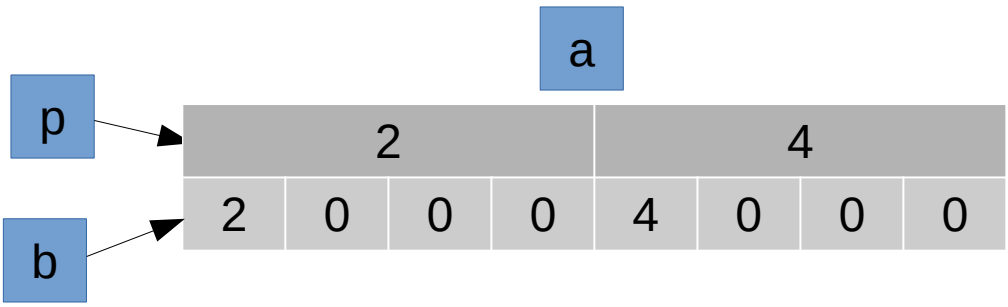


Figura 6: No programa array.cpp, "`int a[2];`" pode ser apontado por ponteiro de int "`int *p; p=a;`" como dois inteiros ou por ponteiro de BYTE "`uint8_t *b; b=(uint8_t*)(a);`" como 8 bytes.

6) Agora, vamos ver como funciona "vector" de C++. As variáveis de C++ são muitas vezes constituídas de duas partes: cabeçalho e dados. Um *vector* de C++ (que seria equivalente a *array* de C) tem essa construção. Veja o programa abaixo, com *array a* e *vector b*, ambos com 100 inteiros com sinal.

```
//vector.cpp
#include <iostream>
#include <vector>
using namespace std;
int main() {
  int a[100];
  cout << "Tamanho de a: " << sizeof(a) << endl;
  vector<int> b(100);
  cout << "Tamanho de b: " << sizeof(b) << endl;
  cout << "Endereco dos dados de b: " << b.data() << endl;
}
```

```
Saída:

Tamanho de a: 400

Tamanho de b: 24
Endereco dos dados de b: 0x55817c1a82c0
```

Usando função *sizeof()*, o programa diz corretamente que array "*int a[100]*" ocupa 4×100=400 bytes. Porém, ele diz que "*vector<int> b(100)*" ocupa apenas 24 bytes! O que está errado? É que o cabeçalho ocupa 24 bytes. Os 100 inteiros (400 bytes) estão no endereço 0x55817c1a82c0, apontado pelo método "*data()*". No cabeçalho, além do apontador *data*, há outras informações, entre eles um apontador para o fim dos dados.

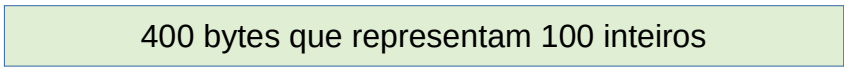


Figura F: Memória alocada para "int a[100];".

Nota: Podemos dar uma espiada no cabeçalho do *b*. Fazendo isso, constatamos que o conteúdo dos primeiros 8 bytes são 0x55817c1a82c0 (o endereço do início dos dados) e o conteúdo dos 8 bytes seguintes são 0x55817c1a8450 (o endereço do fim dos dados). Repare que:
0x55817c1a8450 - 0x55817c1a82c0 = 0x190 = 400 (decimal).

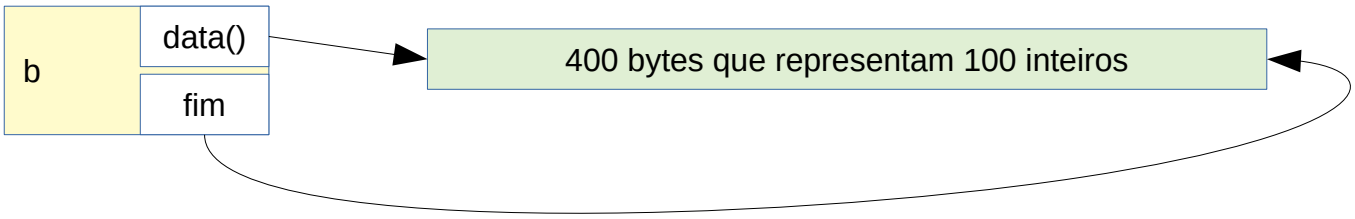


Figura 7: Cabeçalho e dados de "`vector<int> b(100);`"

7) Algo semelhante acontece com *string* de *C* e de *C++*. Apesar de terem o mesmo nome, a estrutura de dados chamada de *string* em *C* é bem diferentes de *string* de *C++* (da mesma forma que *array* de *C* é diferente de *vector* de *C++*). *String* de *C* é uma simplesmente sequência de caracteres (char ou variável de 8 bits com sinal) que termina com o número zero (bits "0000.0000" que pode ser denotado por número 0 ou caracter '\0'). Se executar o comando *char st[] = "CADEIA"* o string *st* fica configurada como segue:

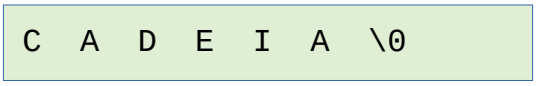


Figura 8: Representação interna da memória após "`char st[] = "CADEIA"`" de C.

Já *string* da linguagem *C++* é composto por um cabeçalho (que inclui, entre outros campos, um apontador para o início do *string*) e uma sequência de caracteres (os dialetos modernos de C++ sempre colocam um '\0' no final de string).

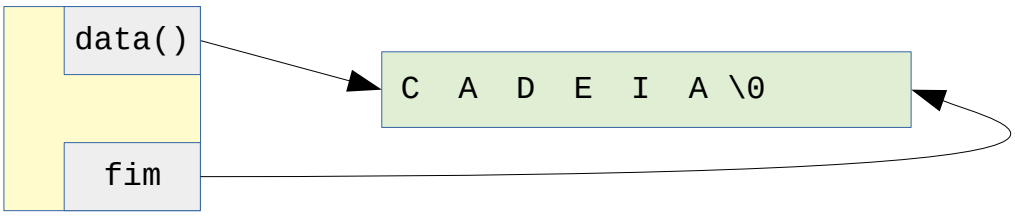


Figura 9: Representação interna da memória após "`st = "CADEIA"`" de C++.

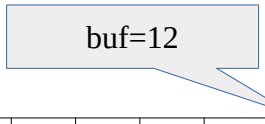
8) Com esse conceito em mente, vamos tentar entender as últimas linhas (marcadas em amarelo) dos programas server5.c e client5.c. Esses programas estão escritas em *C* e portanto usam string estilo *C*. Esses programas usam as seguintes funções:

`strcpy(destino, origem)`: copia string origem para destino, incluindo o caracter '\0'.
`strlen(str)`: calcula o comprimento de string str (sem contar o caracter '\0' final).
`printf("%s\n",str)`: imprime string str que obrigatoriamente deve terminar por '\0'.

Assim, o comando:

```
strcpy(buf,"Me");
```

Copia os três caracteres 'M', 'e' e '\0' para o endereço apontado por buf. Se o endereço apontado por *buf* for 12, teremos o seguinte conteúdo mostrado na figura 10 após esse comando:



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
'l'	'i'	'x'	'o'	'l'	'i'	'x'	'o'	'l'	'i'	'x'	'o'	'M'	'e'	'\0'	'l'	'i'	'x'	'o'	'l'

Figura 10: Configuração da memória após `strcpy(buf,"Me");`

A função `strlen(buf)` retorna 2 ("Me" tem 2 caracteres). Porém, `Buf` na verdade ocupa 3 bytes contando '\n' final.

Se depois executar o comando:

```
if (send(new_fd, buf, strlen(buf)+1, 0) == -1) perror("send");
```

Envia para *socket new_fd*, a partir do endereço *buf*, *strlen(buf)+1=3* bytes (comprimento 2 do *string* “Me” mais um byte de ‘\0’). O último parâmetro representa *flags* e zero indica *flags* padrão.

Se acontecer erro no envio da mensagem, função *send* retorna -1 e imprime uma mensagem de erro. Se conseguiu enviar a mensagem completa ou parcialmente, *send* devolve o número de bytes realmente enviados (que pode ser menos do que o número de bytes pedido para enviar). Como os programas server5.c/client5.c enviam poucos bytes, o autor está ignorando a possibilidade da função *send* enviar menos bytes que o pedido. Mas não podemos fazer isto quando enviarmos uma imagem com muitos bytes.

O comando:

```
if (recv(new_fd, buf, MAXDATASIZE, 0) == -1) perror("recv");
```

Recebe do *socket* new_fd, no máximo MAXDATASIZE bytes e os armazena na memória a partir do endereço *buf*. Se acontecer algum erro na recepção, a função *recv* retorna -1 e imprime uma mensagem de erro. Se recebeu mensagem com sucesso, a função *recv* retorna o número de bytes recebidos (≤MAXDATASIZE). Nos exemplos acima, o autor não usa o número de bytes recebidos a não ser para testar se houve erro, pois sabe que o programa irá receber poucos bytes.

[Durante a aula, explicar o exemplo abaixo.]

Nos programas server5.c e client5.c, o seguinte código cria array de caracteres com comprimento 256:

```
#define MAXDATASIZE 256 // max number of bytes we can get at once
char buf[MAXDATASIZE];
```

No programa server5.c , o seguinte código envia mensagem #1 (incluindo o caractere ‘\0’ final) e fica esperando receber mensagem #2 (de tamanho no máximo 256 bytes):

```
strcpy(buf,"Mensagem #1 do server para client");
if (send(new_fd, buf, strlen(buf)+1, 0) == -1) perror("send");

if (recv(new_fd, buf, MAXDATASIZE, 0) == -1) perror("recv");
printf("server: received '%s'\n",buf);
```

No programa client5.c , o seguinte código espera receber a mensagem #1 (de tamanho no máximo 256 bytes) e envia mensagem #2:

```
if (recv(sockfd, buf, MAXDATASIZE, 0) == -1) perror("recv");
printf("client: received '%s'\n",buf);

strcpy(buf,"Mensagem #2 do client para server");
if (send(sockfd,buf,strlen(buf)+1,0)==-1) perror("send");
```

Linkar com OpenCV

No Cekeikon, eu chamava *uchar8_t* de *GRY* para indicar que é variável usada para armazenar um pixel em níveis de cinza. Similarmente, eu chamava *Vec3b* de *COR* para indicar que é variável usada para armazenar um pixel colorida. Para continuar usando os mesmos nomes, basta fazer mais algumas declarações em *raspberry.hpp:*

```
typedef uint8_t GRY;
#include <opencv2/opencv.hpp>
using namespace cv;
typedef Vec3b COR;
```

Como estamos incluindo o cabeçalho de OpenCV, daqui em diante é necessário linkar com a biblioteca OpenCV:

```
computador$ compila programa -ocv -v3
raspberry$ compila programa -ocv
```

4 Outros métodos *send* e *receive*

Vamos escrever métodos para enviar/recever variáveis de diferentes tipos usando os métodos *sendBytes* e *receiveBytes* que escrevemos acima.

4.1 Métodos *sendUint* e *receiveUint*

Vamos escrever os métodos para enviar/receber inteiros sem sinal de 4 bytes (uint32_t ou unsigned int ou unsigned). Após incluir esses métodos, os headers devem ficar algo como:

```
class SERVER {
  ...
public:
  SERVER();
  ~SERVER();
  void waitConnection();
  void sendBytes(int nBytesToSend, BYTE *buf);
  void receiveBytes(int nBytesToReceive, BYTE *buf);
  void sendUint(uint32_t m);
  void receiveUint(uint32_t& m);
  ...
};
```

```
class CLIENT {
  ...
public:
  CLIENT(string endereco);
  ~CLIENT();
  void sendBytes(int nBytesToSend, BYTE *buf);
  void receiveBytes(int nBytesToReceive, BYTE *buf);
  void sendUint(uint32_t m);
  void receiveUint(uint32_t& m);
  ...
};
```

Nota da aula 3: Para não chamar o professor/especialista repetidamente, chame-os apenas 2 vezes:
1) Quando tiver terminado as lições 1a, 1b, 1c e 1d, chame o professor/especialista e mostre os quatro itens de uma vez (vale 5 pontos os quatro itens).
2) Quando tiver terminado as lições 2a e 2b, chame o professor/especialista e mostre apenas o item 2b (pois 2b inclui 2a, vale 5 os dois itens).

[Lição de casa 1a da aula 3] Usando os novos métodos, faça os seguintes programas funcionarem. Mostre no vídeo que seu server6c/client6c funcionam. Mostre que você usou as funções *ntohl* e *htonl* (veja explicação adiante).

```
//server6c.cpp
#include "projeto.hpp"
int main(void) {
  SERVER server;
  server.waitConnection();
  server.sendUint(1234567890);
  uint32_t u;
  server.receiveUint(u);
  cout << u << endl;
}
```

```
//client6c.cpp
#include "projeto.hpp"
int main(int argc, char *argv[]) {
  if (argc!=2) erro("client6c servidorIpAddr\n");
  CLIENT client(argv[1]);
  uint32_t u;
  client.receiveUint(u);
  cout << u << endl;
  client.sendUint(3333333333);
}
```

Saída esperada:

```
pi@raspberrypi$ server6c
server: Esperando conexao...
server: recebi conexao de 192.168.68.109
3333333333
server: fechei conexao.

usuario@computador$ client6c 192.168.68.110
client: conectando a 192.168.68.110
1234567890
```

Opcional: Classes abstratas, hierarquia de classes e funções virtuais
Nota: Você não é obrigado a fazer este item, mas o seu programa ficará mais limpo se o fizer.

Repare acima que há duas cópias idênticas dos métodos *sendUint* e *receiveUint* nas classes SERVER e CLIENT. Além disso, ambos métodos usam internamente apenas os métodos *sendBytes* e *receiveBytes* das respectivas classes. Num caso como este, é possível usar técnicas de programação orientada a objetos de C++ para eliminar as duplicatas:

(a) Classes abstratas;
(b) Hierarquias de classes;
(c) Funções virtuais.

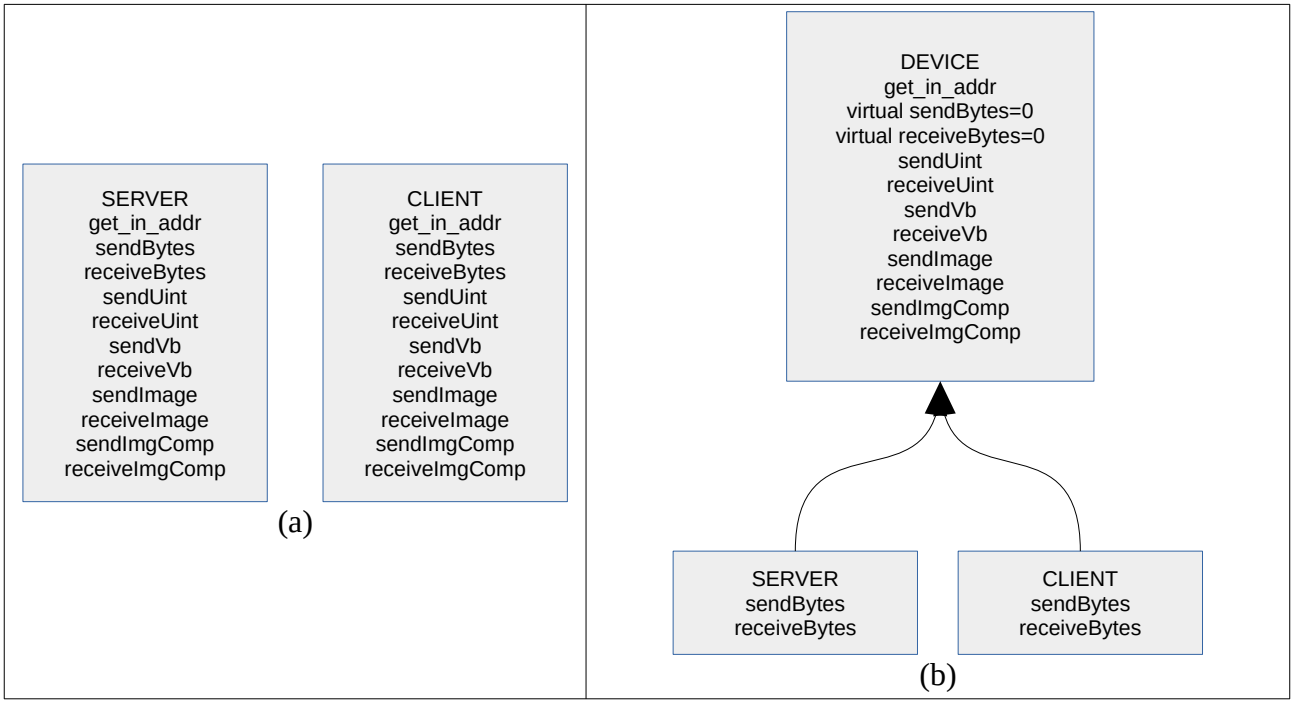


Figura 11: (a) Classes concretas SERVER e CLIENT, onde todos os métodos estão duplicados nas duas classes. (b) Classe abstrata DEVICE e classes derivadas concretas SERVER e CLIENT, sem métodos duplicados (exceto *sendBytes* e *receiveBytes*).

Para isso, vamos criar uma classe abstrata DEVICE da qual derivam as classes SERVER e CLIENT. Na figura acima, as flechas indicam relação de herança.

```
class DEVICE {
public:
  static void *get_in_addr(struct sockaddr *sa);

  virtual void sendBytes(int nBytesToSend, BYTE *buf)=0;
  virtual void receiveBytes(int nBytesToReceive, BYTE *buf)=0;

  void sendUint(uint32_t m);
  void sendVb(const vector<BYTE>& vb);
  void sendImg(const Mat_<COR>& img);
  void sendImgComp(const Mat_<COR>& img);
  void receiveUint(uint32_t& m);
  void receiveVb(vector<BYTE>& vb);
  void receiveImg(Mat_<COR>& img);
  void receiveImgComp(Mat_<COR> &img);
};

class SERVER: public DEVICE {
...

class CLIENT: public DEVICE {
...
```

Uma classe abstrata é um tipo que isola o usuário dos detalhes da implementação. A classe DEVICE é uma classe abstrata, pois as implementações das funções *sendBytes* e *receiveBytes* vão depender de qual classe (SERVER ou CLIENT) você está. As funções *sendBytes* e *receiveBytes* são "puramente virtuais" (pois têm a palavra "virtual" e "=0"), significando que elas devem ser obrigatoriamente especificadas em todas as classes derivadas de DEVICE (veja a seção "3.2.2 Abstract Types" de [Stroustrup2013]). Porém as outras 9 funções podem ser especificadas uma única vez na classe DEVICE que irá funcionar corretamente em ambas as classes SERVER e CLIENT.

Assim, teremos uma função *sendBytes* para SERVER e outra diferente para CLIENT. Da mesma forma, teremos uma função *receiveBytes* para SERVER e outra diferente para CLIENT. As demais funções (*sendUint*, *sendVb*, etc) irão internamente chamar *sendBytes/receiveBytes* do SERVER ou do CLIENT dependendo da classe ao qual pertencem.

Nota: A parte opcional termina aqui.

Big e Little Endian:

Ao escrever os métodos *sendUint* e *receiveUint*, deve-se tomar cuidado com o problema "Big and Little Endian" descrito em [Beej]. O comando "unsigned int m = 0x0a0b0c0d"[2] pode fazer com que o computador armazene o número em dois formatos diferentes internamente:

Tabela 1: Endianness de diferentes processadores.

	byte 0	byte 1	byte 2	byte 3
little endian (x86/Intel/AMD)	0x0d	0x0c	0x0b	0x0a
big endian (Motorola)	0x0a	0x0b	0x0c	0x0d
bi-endian (ARM)	Pode trabalhar como little-endian ou big-endian, mas a maioria dos processadores ARM (incluindo Raspberry) usa little endian.			

A convenção para transmitir números inteiros pela rede é "big endian". Portanto, para trabalhar de acordo com a convenção, devemos converter as variáveis de "little endian" (Intel/Raspberry) para "big endian" antes de transmitir e converter de volta para "little endian" ao receber.

Pode-se converter os números do formato interno do computador (normalmente little endian) para formato de padrão da internet (big endian) ou vice-versa usando as funções do Linux:

```
m=ntohl(m); (net to host long unsigned)
m=htonl(m); (host to net long unsigned)
```

Você pode ler o manual dessas funções digitando no terminal do Linux:

```
$ man ntohl
$ man htonl
```

Na figura 12, há uma variável *m* do tipo "unsigned int" (ou "uint32_t") de 4 bytes com valor 0x0a0b0c0d (decimal 168496141) no endereço 8, armazenado no formato "little endian".

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
4	'e'	'x'	'f'	'a'	3	12	15	0x0d	0x0c	0x0b	0x0a	25	26	27	28	29	30	98	'd'

nBytesToSend=4

`m==0x0a0b0c0d, &m==8`

Figura 12: Uma variável armazenada usando convenção "little endian".

Para transmiti-lo, devemos primeiro convertê-lo para "big endian" (padrão internet):

```
static uint32_t m=0x0a0b0c0d;
m=htonl(m); //host to net long unsigned
```

Aqui, *m* irá valer 0x0a0b0c0d. Depois, devemos chamar a função sendBytes com os parâmetros:

```
sendBytes(4,(BYTE*)&m);
```

Indicando que se deseja enviar 4 bytes a partir do endereço da variável "m" (&m, isto é, o endereço 8). O comando `(BYTE*)` indica ao compilador que interprete o endereço da variável "m" como um apontador para *array* de BYTE (em vez interpretá-lo como apontador para *array* de *uint32_t*).

Com a palavra "static", a variável *m* torna-se "estática" e fica armazenada num endereço fixo que não é destruída até o fim do programa. Sem a palavra "static", a variável *m* seria "local" ou

2 "unsigned int" e "uint32_t" são sinônimos.

“automática”. Isto é, ficaria armazenada na pilha e seria destruída no fim do bloco. Apesar de provavelmente não ser necessário, é mais seguro usar funções *sendBytes* e *receiveBytes* com o *buffer* sempre apontando para uma variável estática, pois assim com certeza a variável não será destruída até o final do programa.

4.2 Métodos sendVb e receiveVb

Digamos que temos um `vector<BYTE> vb` com conteúdo (25, 26, 27, 28, 29, 30). Neste caso, `vb.data()` indica o endereço onde estão armazenados os dados do *vb* (no exemplo, endereço 12); e `vb.size()` indica a quantidade de elementos do vetor.

vb.data()=12; vb.size()=6;

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
4	'e'	'x'	'f'	'a'	3	12	15	0x0d	0x0c	0x0b	0x0a	25	26	27	28	29	30	98	'd'

Figura 13: No *vector<BYTE> vb*, *vb.data()* aponta para o início dos dados e *vb.size()* indica o tamanho do vetor.

Vamos acrescentar mais dois métodos às classes SERVER e CLIENT para poder enviar/receber vetor de bytes. Com isso, os headers devem ficar:

```
class SERVER {
  ...
public:
  SERVER();
  ~SERVER();
  void waitConnection();
  void sendBytes(int nBytesToSend, BYTE *buf);
  void receiveBytes(int nBytesToReceive, BYTE *buf);
  void sendUint(int m);
  void receiveUint(int& m);
  void sendVb(const vector<BYTE>& vb);
  void receiveVb(vector<BYTE>& st);
  ...
};

class CLIENT {
  ...
public:
  CLIENT(string endereco);
  ~CLIENT();
  void sendBytes(int nBytesToSend, BYTE *buf);
  void receiveBytes(int nBytesToReceive, BYTE *buf);
  void sendUint(int m);
  void receiveUint(int& m);
  void sendVb(const vector<BYTE>& vb);
  void receiveVb(vector<BYTE>& st);
  ...
};
```

[Lição de casa 1b da aula 3] Usando os novos métodos e a função *testaVb* abaixo (inclua no projeto.hpp), mostre que os programas server8/client8 abaixo funcionam.

```
bool testaVb(const vector<BYTE> vb, BYTE b) {
  //Testa se todos os bytes de vb possuem valor b
  bool igual=true;
  for (unsigned i=0; i<vb.size(); i++)
    if (vb[i]!=b) { igual=false; break; }
  return igual;
}
```

```
//server8.cpp
//testa sendVb e receiveVb
#include "projeto.hpp"
int main(void) {
  SERVER server;
  server.waitConnection();
  vector<BYTE> vb;

  vb.assign(100000,111);
  server.sendVb(vb);

  server.receiveVb(vb);
  if (testaVb(vb,222)) printf("Recebi corretamente %lu bytes %u\n",vb.size(),222);
  else printf("Erro na recepcao de %lu bytes %u\n",vb.size(),222);

  vb.assign(100000,2);
  server.sendVb(vb);
}
```

```
//client8.cpp
//testa sendVb e receiveVb
#include "projeto.hpp"
int main(int argc, char *argv[]) {
  if (argc!=2) erro("client6 servidorIpAddr\n");
  CLIENT client(argv[1]);
  vector<BYTE> vb;

  client.receiveVb(vb);
  if (testaVb(vb,111)) printf("Recebi corretamente %lu bytes %u\n",vb.size(),111);
  else printf("Erro na recepcao de %lu bytes %u\n",vb.size(),111);

  vb.assign(100000,222);
  client.sendVb(vb);

  client.receiveVb(vb);
  if (testaVb(vb,1)) printf("Recebi corretamente %lu bytes %u\n",vb.size(),1);
  else printf("Erro na recepcao de %lu bytes %u\n",vb.size(),1);
}
```

Para compilar:

```
raspberrypi$ compila server8 -ocv
computador$ compila client8 -ocv -v3
```

Saída esperada:

```
pi@raspberrypi:~/labsist/proj2024/camserver $ server8
server: Esperando conexao...
server: recebi conexao de 192.168.68.109
Recebi corretamente 100000 bytes 222
server: fechei conexao.

hae@Voyager:~/haelinux/haeusp/labsist/proj2024/camserver$ client8 192.168.68.110
client: conectando a 192.168.68.110
Recebi corretamente 100000 bytes 111
```

```
Erro na recepcao de 100000 bytes 1
```

O último “erro” é proposital.

No vector<BYTE> vb:

- *vb.size()* indica o número de elementos do vector,
- *vb.data()* aponta para o endereço onde estão armazenados os dados do vetor,
- *vb.resize(n)* muda o tamanho do vetor para poder armazenar *n* bytes.

Para enviar esta estrutura de dados, envie primeiro o número de elementos do vetor:

```
sendUint(vb.size());
```

Depois, envie `vb.size()` bytes a partir do endereço *vb.data()*:

```
sendBytes(vb.size(),vb.data());
```

Para receber `vector<BYTE>`, receba primeiro o número de bytes do vetor:

```
uint32_t t;
receiveUint(t);
```

Depois, aloque um `vector<BYTE>` com *t* elementos:

```
vector<BYTE> vb;
vb.resize(t);
```

Com isso, construimos o cabeçalho do vetor e alocamos o espaço de *t* bytes para os dados. Finalmente, podemos receber *t* bytes na memória apontada por `vb.data()`:

```
receiveBytes(t,vb.data());
```

Nota: Você pode colocar métodos para enviar/receber outros tipos de dados nas classes SERVER/CLIENT, por exemplo para enviar/receber char, int, string, etc.

5 Visualizar câmera remotamente

5.1 Transmissão de vídeo não-compactado

As imagens de OpenCV também são compostas por cabeçalho e dado.

```
//imagem.cpp
//compila imagem -ocv -v3
#include "projeto.hpp"
int main() {
  Mat_<GRY> a(100,100);
  cout << sizeof(a) << endl;
  printf("%p\n",a.data);
}
```

```
Saída:

96
0xe78410
```

Isto indica que o cabeçalho de *Mat_<GRY> a* ocupa 96 bytes. No cabeçalho, entre outras informações, há a quantidade de linhas e colunas da imagem. Os dados "para valer" da imagem (100×100 bytes) estão na porção da memória que começa no endereço `0xe78410`. Note que aqui se usa o campo *data*, em vez do método *data()*.

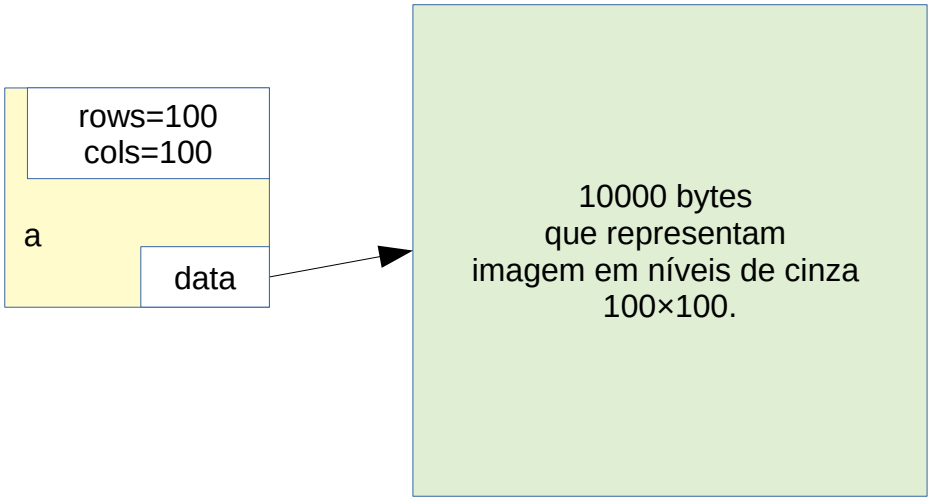


Figura 14: Representação interna de Mat_<GRY> ou Mat_<uchar>.

[Lição de casa 1c da aula 3] Escreva *camclient1.cpp* e *camserver1.cpp*. *Camserver1.cpp* deve capturar os quadros coloridos 480×640 da câmera Raspberry e enviar (sem compressão) para *camclient1.cpp*. *Camclient1.cpp* deve receber os quadros e mostrá-los na tela. Apertando *ESC* no *camclient1*, deve terminar os dois programas.

```
raspberrypi$ camserver1
computador$ camclient1 192.168.0.110
```

Calcule aproximadamente a latência do sistema, isto é, o intervalo de tempo entre algo acontecer e esse acontecimento aparecer na janela do programa no computador. Para mim, deu algo como 2,4s.

Também calcule quadros por segundo (fps). Para mim, deu aproximadamente 3,3 fps. Para calcular fps, você pode usar código semelhante a:

```
int main(int argc, char *argv[]) {
  if (argc!=2) erro("camclient2 ServerIP\n");
  double t1=timeSinceEpoch();
  ...
  int i=0; //Quantidade de quadros recebidos
  while (ch<0) {
    ...
    ch=waitKey(1);
    i++;
  }
  ...
  double t2=timeSinceEpoch();
  double t=t2-t1;
  printf("Quadros=%d tempo=%8.2fs fps=%8.2f\n",i,t,i/t);
}
```

Nota: COR equivale a Vec3b ou Vec<uchar,3> do OpenCV. Consiste de 3 bytes que armazenam a cor no formato BGR.

Sugiro que você acrescente mais dois métodos às classes SERVER e CLIENT para poder enviar/receber imagens coloridas:

```
void sendImg(const Mat_<COR>& img);
void receiveImg(Mat_<COR>& img);
```

No `Mat_<COR> img`:

- *img.isContinuous()* indica se a imagem está armazenada de forma contínua na memória. Quase sempre uma imagem OpenCV é contínua, exceto se estiver usando ROI (região de interesse).
- *img.total()* indica o número total de elementos da imagem (rows*cols).
- *img.data* aponta para o endereço onde estão armazenados o dado da imagem (sequência de bytes).
- *img.create(nl,nc)* aloca espaço para a imagem poder armazenar nl×nc pixels.

Assim, para enviar imagem colorida *img*, você deve primeiro se certificar de que *img* é contínua (será contínua se você não estiver usando ROI). Depois, deve enviar dois inteiros que indicam o tamanho da imagem (*img.rows*, *img.cols*), seguido por 3**img.total()* bytes armazenados no endereço indicado por *img.data*. Usando os métodos que desenvolvemos acima, o código fica:

```
if (img.isContinuous()==false) gera_erro;
sendUint(img.rows); sendUint(img.cols);
sendBytes(img.data,3*img.total());
```

Para receber imagem *img*, você deve primeiro receber dois inteiros (*nl*,*nc*) e depois deve alocar o espaço para armazenar a imagem:

```
  receiveUint(nl); receiveUint(nc);
  Mat_<COR> img(nl,nc);
```

OU

```
  Mat_<COR> img;
  (...)
  receiveUint(nl); receiveUint(nc);
  img.create(nl,nc);
```

A imagem recém-criada é sempre contínua. Por fim, deve receber 3**nl***nc* bytes no endereço apontado por `img.data`.

```
  receiveBytes(img.data,3*img.total())
```

Nota importante: Após receber cada quadro de Raspberry, o computador deve enviar alguma mensagem (por exemplo, o caractere '0') confirmando que recebeu o quadro e que deseja receber mais quadros ou deseja terminar o programa (por exemplo, o caractere 's') quando o usuário tiver apertado ESC. Raspberry deve enviar o quadro seguinte somente após receber confirmação de que computador recebeu o quadro anterior e que quer receber mais quadros. Sem essa confirmação, o sistema pode "travar" depois de algum tempo, pois Raspberry pode tentar enviar mais e mais quadros mesmo que o computador não tenha conseguido recebê-los.

5.2 Transmissão de vídeo compactado como MJPEG

Imagem natural é pouco comprimível, se não informar ao compactador que o que está sendo comprimido é uma imagem e que pode compactar com pequenas perdas de qualidade da imagem.

Abaixo, algumas compressões:

```
   786.447 baboon.ppm - não compactado
   751.044 baboon.zip - compactado sem saber que é imagem
   634.218 baboon.png - compactado sabendo que é imagem, sem perdas
   190.017 baboon.jpg - compactado sabendo que é imagem, com perdas, qualidade 95
    85.428 baboon.jpg - compactado sabendo que é imagem, com perdas, qualidade 80
 3.686.447 flor.ppm - não compactado
 2.767.221 flor.zip - compactado sem saber que é imagem
 1.631.022 flor.png - compactado sabendo que é imagem, sem perdas
   518.059 flor.jpg - compactado sabendo que é imagem, com perdas, qualidade 95
   133.591 flor.jpg - compactado sabendo que é imagem, com perdas, qualidade 80
```

Como se pode ver pela listagem acima:

a) Compressões sem perdas (.zip e .png) conseguem comprimir muito pouco.
b) Compressão com perdas (.jpg) consegue comprimir imagem da ordem de 10 vezes, usando fator de qualidade 80.

[Lição de casa 1d da aula 3] Modifique os programas *camclient1.cpp* e *camserver1.cpp* para criar *camclient2.cpp* e *camserver2.cpp*. A diferença é que agora os quadros devem ser enviados/recebidos com compressão JPEG. *Camserver2.cpp* deve capturar os quadros coloridos 480×640 da câmera Raspberry e enviar (com compressão JPEG) para *camclient2.cpp*. *Camclient2.cpp* deve receber os quadros, descompactá-los e mostrá-los na tela. Apertando *ESC* no *camclient2*, deve parar os dois programas. Mostre que seu camserver2/camclient2 funcionam.

```
raspberrypi$ camserver2
computador$ camclient2 192.168.0.110
```

Calcule aproximadamente a latência do sistema, isto é, o intervalo de tempo entre algo acontecer e esse acontecimento aparecer na janela do programa no computador. Para mim, foi difícil medir a latência, pois o intervalo foi muito pequeno, algo como 0,5s.

Acrescente às classes SERVER e CLIENT do projeto.hpp os métodos que enviam/recebem imagens coloridas compactadas:

```
void sendImgComp(const Mat_<COR>& img);
void receiveImgComp(Mat_<COR>& img);
```

As funções *imencode* e *imdecode* de OpenCV fazem o trabalho de compressão/descompressão JPEG.

imencode: Encodes an image into a memory buffer.

C++: `bool imencode(const string& ext, InputArray img, vector<uchar>& buf, const vector<int>& params=vector<int>())`

Parameters

ext – File extension that defines the output format.
img – Image to be written.
buf – Output buffer resized to fit the compressed image.
params – Format-specific parameters. See imwrite() .

The function compresses the image and stores it in the memory buffer that is resized to fit the result.

O trecho de programa abaixo compacta imagem colorida *image* no formato JPG com qualidade 80 e coloca o vetor de bytes resultante em *vb*.

```
Mat_<COR> image;
vector<BYTE> vb;
vector<int> param{CV_IMWRITE_JPEG_QUALITY,80};
imencode(".jpg",image,vb,param);
```

imdecode: Reads an image from a buffer in memory.
`C++: Mat imdecode(InputArray buf, int flags)`
Parameters
buf – Input array or vector of bytes.
flags – The same flags as in imread() .
The function reads an image from the specified buffer in the memory. If the buffer is too short or contains invalid data, the empty matrix/image is returned. See imread() for the list of supported formats and flags description.

O trecho de programa abaixo decodifica o vetor de bytes recebido.

```
vector<BYTE> vb;
// Função para receber o vetor de bytes vb
Mat_<COR> image=imdecode(vb,1); // Numero 1 indica imagem colorida
```

Consulte o manual do OpenCV para maiores detalhes.

Para enviar uma imagem *Mat_<COR> img*, você deve codificá-lo transformando num *vector<BYTE> vb*. Depois, transmite *vb*. Recebendo *vb*, você descompacta-o para obter novamente a imagem colorida.

Utilizando esta solução, no meu sistema chegou-se a 18 fps 480×640. No ambiente VM dentro de Windows, a transmissão também teve a mesma velocidade de 18 fps.

Nota importante: Após receber cada quadro de Raspberry, o computador deve enviar alguma mensagem confirmando que recebeu o quadro e deseja receber mais quadros ou quer fechar o sistema (o usuário apertou ESC). Sem essa confirmação, o sistema pode parar de funcionar depois de algum tempo.

Observação: É possível compactar imagem e transmitir imagem em paralelo, usando dois núcleos da Raspberry. Porém, desconfio que isso melhoraria *fps* mas pioraria latência, pois pode ser que o quadro sendo compactado não seja o último quadro disponível. Quem quiser, pode implementar e verificar.

6 Fases 1 e 2 do projeto

6.1 Fase 1 - Transmissão de vídeo e comandos

Neste fase, vamos criar uma arquitetura servidor-cliente TCP/IP local via roteador wifi, sem controlar os motores. Faça um programa servidor1.cpp para rodar na Raspberry e outro programa cliente1.cpp para rodar no computador.

Estando Raspberry e computador ligados no mesmo roteador, a forma de chamar os dois programas deve ser:

```
raspberrypi$ servidor1
computador$ cliente1 192.168.0.110 [videosaida.avi] [t/c]
```

onde:

- 192.168.0.110 é o endereço IP da Raspberry.
- *Videosaida.avi* é o nome do vídeo onde as imagens mostradas na tela ou capturadas pela câmera serão armazenadas. Se o vídeo de saída não for especificado, o conteúdo da tela não será gravado.
- O argumento *t* ou *c* indica respectivamente gravar uma cópia da tela (com teclado virtual - default) ou somente a imagem capturada pela câmera. Usando o argumento *t*, quadros como a figura 15 serão gravados. Usando o argumento *c*, serão gravados somente a imagem capturada pela câmera (a parte direita da figura 15).

Nota: Em 2024, não foi obrigatório fazer o exercício da apostila raspicam_wiringpi, pg. 7, que grava o vídeo capturado num arquivo. Leia o enunciado desse exercício que explica como salvar o vídeo num arquivo.
Nota: Em 2017, os comandos do computador eram inseridos pelo teclado numérico. O problema é que a interface HighGUI (High-level Graphical User Interface) do OpenCV permite descobrir quando uma tecla foi pressionada, mas não informa o instante em que essa tecla foi solta. Assim, apertando (por exemplo) a tecla 8 para ir para frente, não era possível saber quando o carrinho deveria parar, pois não dava para saber quando a tecla 8 foi solta. Usar uma biblioteca GUI mais poderosa complicaria o projeto. Por outro lado, HighGUI consegue controlar melhor o mouse: permite determinar se um botão do mouse está apertado ou solto e em que posição (x, y) da janela está o ponteiro do mouse. A partir de 2018, construimos um "teclado numérico virtual" para ser apertado com o mouse.

Vamos construir um "teclado numérico virtual", para ser apertado com o mouse, com os seguintes botões virtuais (figura 15):

Virar à esquerda (equivale à tecla numérica 7)
Ir para frente (8)
Virar à direita (9)
Virar acentuadamente à esquerda (4)
Não faz nada (5)
Virar acentuadamente à direita (6)
Virar à esquerda dando ré (1)
Dar ré (2)
Virar à direita dando ré (3)

Use a tecla física ESC para sair do programa.

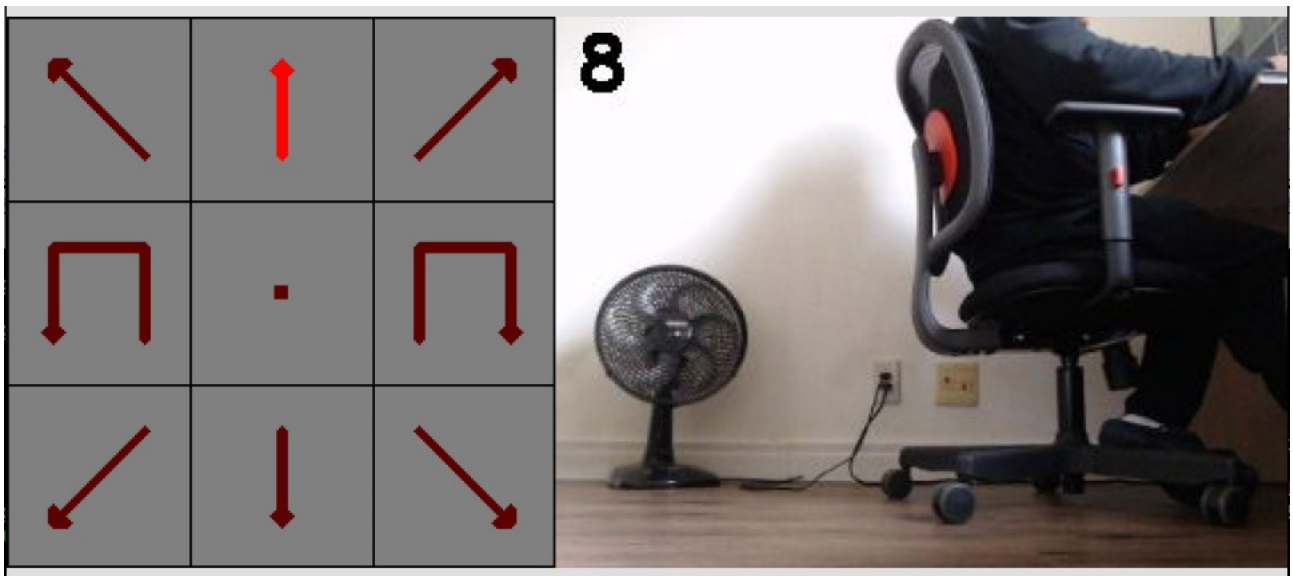


Figura 15: Teclado numérico virtual para controlar carrinho com tecla “para frente” apertado. A tecla “para frente” corresponde ao teclado numérico “8”.

Nesta fase, o programa não vai controlar os motores. O funcionamento do sistema deve ser:

1) O usuário inicia o programa *servidor1* no Raspberry e espera receber conexão do computador.
2) O usuário inicia o programa *cliente1* no computador, fornecendo o número IP do Raspberry. O computador envia uma mensagem (por exemplo, “0”) ao Raspberry para dizer que está pronto para receber uma imagem.
3) Raspberry recebe a mensagem inicial, captura uma imagem colorida 240×320 da câmera, compacta-a e envia-a ao computador.
4) Computador recebe a imagem compactada, descompacta-a, “gruda-a” ao teclado virtual horizontalmente e mostra-a na janela (e salva-a no vídeo se o vídeo de saída tiver sido especificado).
5) O usuário visualiza a imagem grudada ao teclado virtual numa janela do computador. Se o usuário apertou ESC, o computador envia uma mensagem (por exemplo, “s”) para terminar o programa do Raspberry e finaliza o programa do computador. Caso contrário, o computador verifica se o usuário apertou algum botão virtual (1 a 9). Nesse caso, o computador envia o código do botão virtual apertado (“1” a “9”). Se nenhum botão virtual tiver sido apertado, o computador envia uma mensagem específica (por exemplo, “0”).
6) Raspberry recebe a mensagem (“s” para terminar o programa, “0” para indicar não-pressionamento do botão, “1” a “9” para indicar o pressionamento do botão). Se a mensagem for “s”, termina o programa. Caso contrário, Raspberry captura uma nova imagem colorida 240×320 da câmera e insere a mensagem recebida nessa imagem. Depois, compacta-a e envia-a ao computador. A mensagem é inserida na imagem para mostrar que Raspberry recebeu corretamente o comando do computador. Na figura 1, foi selecionada “ir para frente” correspondente à tecla “8”.
7) Repete a partir da linha (4).

Nota: Neste sistema, podemos entender por que TCP/IP não entope a rede com mensagens quando a rede estiver lenta. Repare que Raspberry só vai enviar um novo quadro (linha 3 ou 6) quando receber uma mensagem do computador. Por sua vez, o computador só vai enviar a mensagem informando que está pronto quando receber um quadro do Raspberry (linha 4). Assim, se a rede estiver lenta, a quantidade de comunicações entre computador e Raspberry vai diminuir automaticamente.

Nota: Pode usar funções não-documentadas de OpenCV *hconcat* e *vconcat* para grudar duas imagens:

https://stackoverflow.com/questions/11019272/concatenate-mat-in-opencv

```
Mat A, B;
... //In this part you initialize the Mat A and Mat B.
Mat H, V; //These are the destination matrices
hconcat(A, B, H);
vconcat(A, B, V);
```

[Lição de casa 2a da aula 3] Mostre que seu servidor1/cliente1 está funcionando. Mostre também que a gravação de vídeo funciona.

Ajuda à lição de casa 2a: O programa highgui.cpp abaixo é um exemplo do uso do mouse dentro do HighGUI. Há mais exemplos na apostila "highgui" (PDF, DOC) e no manual do OpenCV.

```
 1 // highgui.cpp
 2 // computador$ compila highgui -ocv -v3
 3 #include "../projeto.hpp"
 4
 5 int estado=0; //0=nao_apertado, 1=apertou_botao_1 2=apertou_botao_2
 6 void on_mouse(int event, int c, int l, int flags, void* userdata) {
 7   if (event==EVENT_LBUTTONDOWN) {
 8     if      (  0<=l && l<100 &&   0<=c && c<100 ) estado=1;
 9     else if (  0<=l && l<100 && 100<=c && c<200 ) estado=2;
10     else estado=0;
11   } else if (event==EVENT_LBUTTONUP) {
12     estado=0;
13   }
14 }
15
16 int main() {
17   COR cinza(128,128,128);
18   COR vermelho(0,0,255);
19   Mat_<COR> imagem(100,200,cinza);
20   namedWindow("janela",WINDOW_NORMAL);
21   resizeWindow("janela",2*imagem.cols,2*imagem.rows);
22   setMouseCallback("janela", on_mouse);
23   imshow("janela",imagem);
24   while ((waitKey(1)&0xff)!=27) { // ESC=27 sai do programa
25     imagem.setTo(cinza);
26     if (estado==1) {
27       for (int l=0; l<100; l++)
28         for (int c=0; c<100; c++)
29           imagem(l,c)=vermelho;
30     } else if (estado==2) {
31       for (int l=0; l<100; l++)
32         for (int c=100; c<200; c++)
33           imagem(l,c)=vermelho;
34     }
35     imshow("janela",imagem);
36   }
37 }
```

Programa highgui.cpp

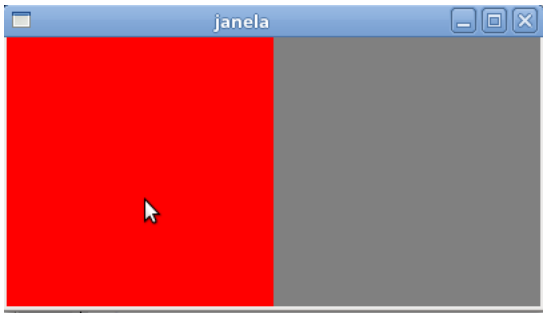


Figura 16: Janela do programa highgui.cpp que implementa dois botões controlados por mouse.

Explicação da função on_mouse: A função “on_mouse” (linha 5 do programa highgui.cpp) é uma função “callback”. Ela fica associada à janela chamada “janela” pelo comando abaixo:

```
setMouseCallback("janela", on_mouse);
```

Após este comando, toda vez que acontecer algum evento envolvendo mouse dentro da “janela”, a função “on_mouse” será chamada. Estes eventos incluem: apertar botão do mouse, soltar botão do mouse, mover o cursor do mouse dentro da janela, etc.

Explicação da função waitKey: Copio abaixo a descrição da função *waitKey* do manual do OpenCV. Essa função é essencial para o funcionamento do HighGUI e deve ser chamada periodicamente (caso contrário, a imagem não será mostrada na janela):

```
waitKey: Waits for a pressed key.
int waitKey(int delay=0)
Parameters: delay – Delay in milliseconds. 0 is the special value that means “forever”.
```

The function waitKey waits for a key event infinitely (when delay ≤ 0) or for delay milliseconds, when it is positive. Since the OS has a minimum time between switching threads, the function will not wait exactly delay ms, it will wait at least delay ms, depending on what else is running on your computer at that time. It returns the code of the pressed key or -1 if no key was pressed before the specified time had elapsed.

Note: This function is the only method in HighGUI that can fetch and handle events, so it needs to be called periodically for normal event processing unless HighGUI is used within an environment that takes care of event processing.

Note: The function only works if there is at least one HighGUI window created and the window is active. If there are several HighGUI windows, any of them can be active.

Ajuda à lição de casa 2a: O programa abaixo mostra como colocar letra/número e como desenhar retas/flechas na imagem, usando rotinas de OpenCV.

```
// desenha.cpp
// computador$ compila desenha -ocv -v3
#include "../projeto.hpp"
int main() {
  Mat_<COR> a(100,100,COR(255,255,255));
  putText(a, "XY", Point(10,50), 0, 0.8, Scalar(0,0,0), 2);
  line(a, Point(70,10), Point(10,90), Scalar(255,0,0), 2);
  arrowedLine(a, Point(90,10), Point(30,90), Scalar(0,0,255));
  namedWindow("janela");
  imshow("janela",a);
  waitKey(0);
  imwrite("desenha.png",a);
}
```

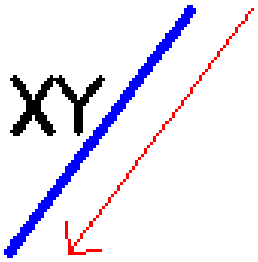


Figura 17: Saída do programa desenha.cpp (desenha.png)

6.2 Fase 2 - Controle remoto manual do carrinho com transmissão de vídeo

Neste fase, vamos controlar manualmente o carrinho à distância, onde o usuário vai enxergar na tela do computador o vídeo capturado pelo raspberry e vai comandar manualmente os motores do carrinho apertando as teclas virtuais do computador especificadas na fase 1.

Modifique o programa servidor1.cpp para criar o programa servidor2.cpp. O programa servidor2.cpp deve efetivamente controlar os dois motores do carrinho (em vez de apenas mostrar o estado virtual dos motores no vídeo). Escreva o programa cliente2.cpp correspondente para rodar no computador. Como antes, a sintaxe para chamar os programas é:

```
raspberry$ servidor2
computador$ cliente2 192.168.0.110 [videosaida.avi] [t/c]
```

Para compilar um programa no Raspberry e linkar com WiringPi (além de OpenCV), você deve usar a opção "-w":

```
raspberry$ compila servidor2 -ocv -w
computador$ compila cliente2 -ocv -v3
```

Além disso, é necessário incluir os cabeçalhos das funções de wiringPi em servidor2.cpp

```
//servidor2.cpp
//raspberry$ compila servidor2 -ocv -w
#include "../projeto.hpp"
#include <wiringPi.h>
#include <softPwm.h>
...
```

O vídeo seguinte mostra o controle manual do carrinho: controlemanual4.avi

O funcionamento da fase 2 é praticamente idêntico à fase 1. A única diferença é o item 6, nos trechos marcados em amarelo.

1) O usuário inicia o programa *servidor1* no Raspberry e espera receber conexão do computador.
2) O usuário inicia o programa *cliente1* no computador, fornecendo o número IP do Raspberry. O computador envia uma mensagem (por exemplo, "0") ao Raspberry para dizer que está pronto para receber uma imagem.
3) Raspberry recebe a mensagem e captura uma imagem colorida 240×320 da câmera, compacta-a e envia-a ao computador.
4) Computador recebe a imagem compactada, descompacta-a, "gruda-a" ao teclado virtual horizontalmente e mostra-a na janela (e salva-a no vídeo se o vídeo de saída tiver sido especificado).
5) O usuário visualiza a imagem grudada ao teclado virtual numa janela do computador. Se o usuário apertou ESC, o computador envia uma mensagem (por exemplo, "s") para terminar o programa do Raspberry e finaliza o programa do computador. Caso contrário, o computador verifica se o usuário apertou algum botão virtual (1 a 9). Nesse caso, o computador envia o código do botão virtual apertado ("1" a "9"). Se nenhum botão virtual tiver sido apertado, o computador envia uma mensagem específica (por exemplo, "0").
6) Raspberry recebe a mensagem ("s" para terminar o programa, "0" para indicar não-pressionamento do botão, "1" a "9" para indicar o pressionamento do botão). Se a mensagem for "s", termina o programa. Caso contrário, movimenta os motores de acordo com a mensagem recebida. Depois, Raspberry captura uma nova imagem colorida 240×320

da câmera. Não insere a mensagem recebida no nessa imagem, para que possamos utilizar o vídeo resultante para treinar rede neural convolucional, a ser usado nos próximos anos. Depois, compacta a imagem e envia-a ao computador.

7) Repete a partir da linha (4).

Calibre as potências (PWM) dos dois motores de forma que o carrinho ande para frente en linha reta quando apertar flecha para cima (tecla virtual “8”) e ande para trás em linha reta quando apertar a flecha para baixo (tecla virtual “2”).

Nota: Estou obtendo ≈26 fps.

[Lição de casa 2b da aula 3] Mostre que servidor2/cliente2 está funcionando.

[Aula 3. Fim.]

Referências

[Stroustrup2013] Stroustrup, Bjarne. *The C++ programming language* - Fourth edition. 2013 Pearson Education.