Programação de Rede TCP/IP

Para esta aula, vocês devem ler antes da aula o livro "Beej's Guide to Network Programming Using Internet Sockets", disponível gratuitamente em:

https://beej.us/guide/bgnet/ (acessado em 27/09/2018)

Além disso (evidentemente) devem ler esta apostila.

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).

A partir desta aula, vocês vão desenvolver programas. Para não ter perigo de perder os programas desenvolvidos, tirem vários 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) e/ou na nuvem.

1 Comunicação TCP/IP

Nas fases 1 e 2 do projeto, vamos fazer um programa que controla o carrinho remotamente em tempo real. Isto é, que permite que o usuário visualize num computador remoto os quadros capturados pela câmera da Raspberry e controlar manualmente (a partir do computador) os motores do carrinho. A primeira ideia seria fazer um programa local em Raspberry e executar esse programa remotamente usando "VNC" ou "ssh -X" ou "ssh -X -C". Seria uma solução bem simples e direta.

Para verificar que isto não funciona, tente executar o programa cekraspicam2.cpp (apostila raspicam_wiringpi.odt) usando "VNC" ou "ssh -X" ou "ssh -X -C". Vejam que há muito atraso na transmissão de imagens (de tamanho 480x640). No meu sistema:

ssh -X: 3.96 fps

ssh -X -C: 3.77 fps (com compressão fica pior, se não sabe que o que está sendo transmitido são imagens).

VNC: O programa informa 29.50 fps. Porém isto é fps no Raspberry. Desses quadros, uma boa parte não chega ao computador.

Tente reduzir a resolução para 360x480 ou 240x320 e/ou tente desligar a criptografia do VNC. Melhora um pouco, mas não a ponto de permitir controlar o carrinho em tempo real.

Para descobrir por que não funciona, vamos verificar a velocidade real de transmissão de dados pelo wifi. Para medir a velocidade de conexão entre dois computadores, pode-se usar o comando iperf:

```
sudo apt-get install iperf (no Raspberry e no computador)
iperf -s (no servidor, isto é, Raspberry)
iperf -c 192.168.0.110 (no computador cliente)
```

onde 192.168.0.110 é o endereço IP do Raspberry. Fazendo isto na minha instalação, dá apenas 13Mbits/s, muito menos do que a suposta velocidade do roteador de 150 ou 300 Mbits/s. Para transmitir 30 imagens coloridas 640x480 por segundo sem compressão, precisaria de uma comunicação de 221Mbits/s. Em 13MBits/s passa menos de 2 quadros por segundo.

Nota: Fazendo a ligação roteador-computador com fio ethernet, obtive 38Mbits/s. Fazendo as duas ligações com fio ethernet (roteador-computador e roteador-raspberry), obtive 94Mbits/s.

Qual é a solução? Fazer um programa próprio para transmitir vídeos, com compactação quadro-a-quadro por JPG. Escolhendo uma taxa de compressão adequada, a compressão reduzirá em aproximadamente 10 vezes a quantidade de dados a ser transmitida. Além disso, este esquema permite que o processamento de visão computacional (fases 3-6 do projeto) seja feito no computador remoto. Utilizando esta solução, no meu sistema chegou-se a 11.81 fps.

Para fazer transmissão-recepção 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. [particularidade de BashWin] Nota: Se você está usando BashWin, rode talker no BashWin e listener no Raspberry, pois BashWin não funciona como listener. Opcional: Modifique talker-listener para transmitir os quadros do vídeo capturado - verifique como este protocolo perde pacotes e portanto não dá para usá-lo na diretamente no nosso projeto.

3) No capítulo 6, há um exemplo de conexão TCP/IP server-client. Baixe os programas, compile-os e execute-os. Nota: Se você está usando BashWin, rode client no BashWin e server no Raspberry, pois BashWin não funciona como server.

Iremos modificar server.c e client.c para transmitir o vídeo capturado pela Raspberry. Para facilitar um pouco 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. Modifiquei-o para servir um único cliente.
b) Modifiquei os programas server.c e client.c para tanto enviar como receber dados, para deixar o exemplo mais completo.

Compile e execute server5.c/client5.c:

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

OU

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

O endereço IP 127.0.0.1 é o "loopback".

```
/*
** 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
```

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

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

2 Classes SERVER e CLIENT

Os programas server5.c/client5.c acima têm dois problemas. O primeiro problema 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 não foram enviados. O mesmo acontece com a função recv - pode receber menos bytes do que o solicitado. Essas funções retornam -1 em caso de erro. Você pode consultar o manual dessas funções escrevendo no terminal do Linux:

```
>man send
>man recv
```

O segundo problema é que os programas server5.c/client5.c (escritos na linguagem C) estão "uma bagunça". 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 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 e não mais nos preocuparmos com o funcionamento interno. Vamos escrever as classes SERVER e CLIENT em várias etapas.

2.1 Métodos sendBytes e receiveBytes

Escreveremos em primeiro lugar os métodos para mandar/receber *n* bytes (e não para mandar/receber qualquer outro tipo de dado) porque fica muito simples mandar/receber qualquer outro tipo de variável chamando internamente sendBytes e receiveBytes.

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);
  ...
};
```

Nota: "BYTE" é o mesmo que "unsigned char" ou "uchar8_t" ou "uchar" ou "GRY". É o termo usado pelos programas da Microsoft e que adotei no Cekeikon. Se não quiser incluir Cekeikon, basta escrever no começo do seu programa: `typedef uint8_t BYTE;`

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 nesse código, vai ter que alterar todas as cópias dela, propiciando introdução de erros. Como há vários programas que vão usar as classes SERVER e CLIENT, é recomendável deixar o código num único lugar. Para isso, vamos guardar essas classes num arquivo "include" chamado "projeto.hpp" e incluir esse arquivo em todos os programas que vão utilizá-las. Também coloque em projeto.hpp a função testaBytes abaixo (testa se todos os n bytes da memória apontada por buf possuem o valor b):

```
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;
}
```

Escrevendo corretamente essas duas classes e a função testaBytes no arquivo projeto.hpp, os programas abaixo devem funcionar:

```
//server6b.cpp
//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);
  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
//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);
  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);
}
```

Exercício 1 valendo 2 pontos) 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 ao professor server6b/client6b funcionando. (Para não ficar chamando professor toda hora, sugiro que chamem-no quando terminarem os exercícios 1, 2 e 3 e mostrem os três exercícios de uma só vez).

Saída esperada:

```
pi@raspberrypi ~/labinteg/projeto/camserver $ server6b
server: got connection from 192.168.0.109
Recebeu corretamente 100000 bytes 214

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

2.2 Métodos sendUint e receiveUint

Agora, vamos escrever métodos para enviar/recever variáveis inteiras sem sinal de 4 bytes (uint32_t ou unsigned int ou unsigned). Os headers (após incluir esses métodos) devem ser 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);
  ...
};
```

Usando os novos métodos, faça os seguintes programas funcionarem:

```
//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("client6b servidorIpAddr\n");
  CLIENT client(argv[1]);
  uint32_t u;
  client.receiveUint(u);
  cout << u << endl;
  client.sendUint(3333333333);
}
```

Nota: Ao escrever os métodos sendInt e receiveInt, tome cuidado com o problema "Big and Little Endian" descrito em [Beej]. O comando "unsigned m = 0x0a0b0c0d" 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 (Intel)	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) usam little endian.			

Há uma convenção universal para transmitir números inteiros pela rede. Pode-se converter para os números no formato do computador para formato de rede usando as funções do Linux:

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

Pode olhar o manual dessas funções digitando:

```
>man ntohl
```

O programa endian2.cpp foi executado em Raspberry, obtendo a saída listada abaixo do programa.

```
//endian2.cpp
#include <cstdio>
#include <arpa/inet.h>

int main() {
  uint32_t i=0x0a0b0c0d;
  uint8_t* pt=(uint8_t*)(&i);
  printf("Formato interno: ");
  printf("%02x ",pt[0]);
  printf("%02x ",pt[1]);
  printf("%02x ",pt[2]);
  printf("%02x ",pt[3]);
  printf("\n");
  i=htonl(i);
  printf("Formato da rede: ");
  printf("%02x ",pt[0]);
  printf("%02x ",pt[1]);
  printf("%02x ",pt[2]);
  printf("%02x ",pt[3]);
  printf("\n");
}
```

```
Formato interno: 0d 0c 0b 0a (tanto Intel como Raspberry)
Formato da rede: 0a 0b 0c 0d
```

Isto mostra que Raspberry (assim como Intel) usa formato little endian mas o formato da rede é big endian. Assim, você deve converter little endian para big endian antes de transmitir variável unsigned int (int32_t) e deve converter big endian para little endian quando recebe.

Exercício 2 valendo 2 pontos) Mostre ao professor server6c/client6c funcionando. Mostre que você usou as funções ntohl e htonl.

Saída esperada:

```
pi@raspberrypi:~/labinteg/projeto/camserver $ server6c
server: got connection from 192.168.0.100
3333333333

hae@Royale ~/labinteg/projeto/camserver $ client6c 192.168.0.110
client: connecting to 192.168.0.110
1234567890
```

2.3 Métodos sendString e receiveString

Este item não é obrigatório (não vale nota).

Acrescente os métodos sendString e receiveString às classes SERVER e CLIENT. 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);
  void sendInt(int m);
  void receiveInt(int& m);
  void sendString(const string& st);
  void receiveString(string& st);
  ...
};

class CLIENT {
  ...
public:
  CLIENT(string endereco);
  ~CLIENT();
  void sendBytes(int nBytesToSend, BYTE *buf);
  void receiveBytes(int nBytesToReceive, BYTE *buf);
  void sendInt(int m);
  void receiveInt(int& m);
  void sendString(const string& st);
  void receiveString(string& st);
  ...
};
```

Usando os novos métodos dessas duas classes, podemos escrever os programas server7.cpp/client7.cpp que fazem o mesmo que os programas server5.c/client5.c como:

```
//server7.cpp
(insira aqui os includes e a classe SERVER)
int main(void) {
  SERVER server;
  server.waitConnection();
  server.sendString("Mensagem #1 do server para client");
  string st;
  server.receiveString(st);
  cout << st << endl;
  server.sendString("Mensagem #3 do server para client");
}

//client7.cpp
(insira aqui os includes e a classe CLIENT)
int main(int argc, char *argv[]) {
  if (argc!=2) erro("client7 servidorIpAddr\n");
  CLIENT client(argv[1]);
  string st;
  client.receiveString(st);
  cout << st << endl;
  client.sendString("Mensagem #2 do client para server");
  client.receiveString(st);
  cout << st << endl;
}
```

Nota: O método st.length() indica número de caracteres do string st. O método st.data() aponta o endereço onde os caracteres do string st estão armazenados. Assim, para enviar o string st, você

deve enviar o comprimento do string st.length() seguido por st.length() bytes no endereço indicado por st.data().

Nota: O método st.resize(n) muda o tamanho do string st para poder armazenar n caracteres. Assim, para receber n caracteres dentro de string st, você deve primeiro receber um inteiro n (o tamanho do string). Depois, deve alocar o espaço para poder receber n caracteres:

```
    string st;
    st.resize(n);
```

Por fim, deve receber n bytes no endereço apontado por st.data().

2.4 Métodos sendVb e receiveVb

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 sendInt(int m);
  void receiveInt(int& m);
  void sendString(const string& st); //nao-obrigatorio
  void receiveString(string& st); //nao-obrigatorio
  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 sendInt(int m);
  void receiveInt(int& m);
  void sendString(const string& st); //nao-obrigatorio
  void receiveString(string& st); //nao-obrigatorio
  void sendVb(const vector<BYTE>& vb);
  void receiveVb(vector<BYTE>& st);
  ...
};
```

Usando os novos métodos, e a função abaixo (a ser incluída no projeto.hpp), os programas server8/client8 devem funcionar:

```
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);
}
```

No vector<BYTE> vb:

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

Assim, para enviar vb, você deve enviar o tamanho do vetor vb.size() seguido por vb.size() bytes armazenados no endereço indicado por vb.data(). Para receber n bytes e armazenar em vb, você deve primeiro receber um inteiro n, e depois deve alocar o espaço para poder receber n bytes:

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

Por fim, deve receber n bytes no endereço apontado por vb.data().

Exercício 3 valendo 2 pontos) Mostre ao professor server8/client8 funcionando.

Saída esperada:

```
pi@raspberrypi:~/labinteg/projeto/camserver $ server8
server: got connection from 192.168.0.100
Recebi corretamente 100000 bytes 222

hae@Royale ~/labinteg/projeto/camserver $ client8 192.168.0.110
client: connecting to 192.168.0.110
Recebi corretamente 100000 bytes 111
Erro na recepcao de 100000 bytes 1
```

Nota: Se achar necessário, você pode colocar métodos para enviar/receber outros tipos de dados nas classes SERVER/CLIENT.

3 Visualizar câmera remotamente

3.1 Transmissão de vídeo descompactado

Escreva camclient1.cpp e camserver1.cpp. Camserver1.cpp deve capturar os quadros coloridos 480x640 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 parar os dois programas.

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

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. Normalmente, uma imagem 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 x nc pixels.

Assim, para enviar img, você deve primeiro se certificar de que img é contínuo (será contínuo se você não está usando ROI). Enviar o tamanho da imagem (img.rows, img.cols), seguido por 3*img.total() bytes armazenados no endereço indicado por img.data(). Para receber imagem img, você deve primeiro receber dois inteiros (nl,nc) e depois deve alocar o espaço para armazenar a imagem:

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

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

Nota: Após receber cada quadro de Raspberry, o computador deve enviar alguma mensagem confirmando que recebeu o quadro. Sem essa confirmação, o sistema pode parar de funcionar depois de algum tempo.

Exercício 4 valendo 2 pontos) Mostre ao professor camserver1/camclient1 funcionando. (Para não ficar chamando professor toda hora, sugiro que chamem-no quando terminarem os exercícios 4 e 5 e mostrem os dois exercícios de uma só vez).

3.2 Transmissão de vídeo compactado

Imagem natural é pouco comprimível, se não informar ao compactador que o que está sendo comprimido é uma imagem e permitir compactação com perdas. 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
```

Compressão com perdas (jpg) consegue comprimir uma imagem da ordem de 10 vezes, com qualidade 80.

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 480x640 da câmera Raspberry e enviar (com compressão JPEG) para camclient2.cpp. Camclient2.cpp deve receber os quadros e mostrá-los na tela. Apertando ESC no camclient2, deve parar os dois programas.

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

Você pode acrescentar a classes SERVER e CLIENT do projeto.hpp os métodos:

```
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. Repare que uchar, unsigned char, BYTE, GRY, uint8_t são todos sinônimos.

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 a 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 imagem colorida.

Nota: Após receber cada quadro de Raspberry, o computador deve enviar alguma mensagem confirmando que recebeu o quadro. Sem essa confirmação, o sistema pode parar de funcionar depois de algum tempo.

Exercício 5 valendo 2 pontos) Mostre ao professor camserver2/camclient2 funcionando. Mostre que os quadros estão sendo enviados compactados. Verifique como a versão com compressão é mais "responsivo" do que a versão sem compressão.

4 Fases 1 e 2 do projeto

4.1 Transmissão de vídeo pela Raspberry e transmissão de comandos manuais de controle pelo computador (fase 1)

Neste fase, vamos criar uma arquitetura servidor-cliente TCP/IP local via roteador wifi. Nesta fase, não vai haver controle dos 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]
```

onde 192.168.0.110 (por exemplo) é o endereço IP da Raspberry e videosaida.avi é um arquivo de vídeo (opcional) onde as imagens mostradas na tela serão armazenadas. Se o vídeo de saída não for especificada, o conteúdo da tela não será armazenada.

Servidor1 captura vídeo (colorido, 240x320) da câmera e o transmite em tempo real através do wifi do roteador para o computador. Use a compressão JPEG quadro a quadro para diminuir a quantidade de dados a serem transmitidos (sem compressão, a taxa quadros por segundo poderá ser baixa e o vídeo chegar com atraso).

No ano passado (2017), os comandos do computador eram inseridos pelo teclado, por exemplo tecla 7 = virar à esquerda, tecla 8 = ir para frente, etc. O problema é que a interface HighGUI (High-level Graphical User Interface) do OpenCV envia um caractere quando uma tecla é pressionada, mas não informa o instante em que essa tecla é solta. Assim, apertando (por exemplo) a tecla 8 para ir para frente, não é possível saber quando o carrinho deve parar (pois não dá para saber quando a tecla 8 foi solta). A solução adotada (ruim) era apertar uma outra tecla para parar o carrinho. Poderíamos resolver este problema usando uma outra biblioteca GUI, porém o projeto ficaria bem mais complexo. Por outro lado, HighGUI permite um controle melhor do 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. Assim, este ano (2018), vamos construir um "teclado virtual" (correspondente ao teclado numérico do computador) para ser apertado com mouse, com os seguintes botões virtuais (figura 1):

Virar à esquerda (equivaleria à tecla numérica 7)
Ir para frente (8)
Virar à direita (9)
Virar acentuadamente à esquerda (4)
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.

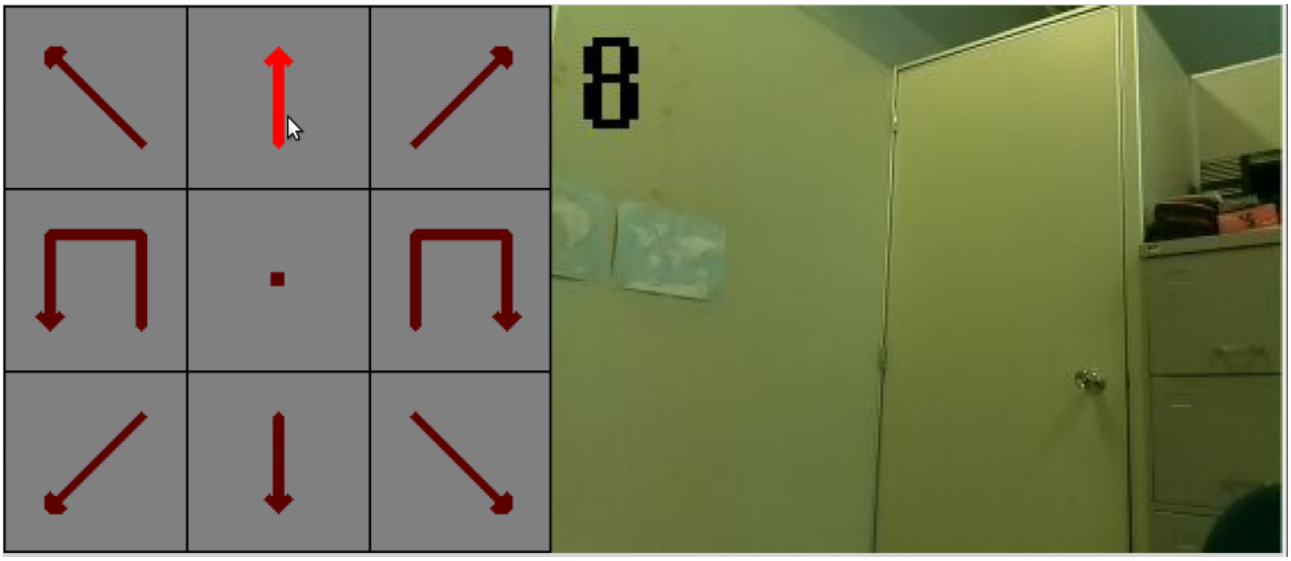

Figura 1: Teclado virtual para controlar carrinho.

Nesta fase, não vai haver controle dos motores. O funcionamento do sistema deve ser: Raspberry captura imagens da câmera, compacta e envia ao computador. Computador recebe as imagens da câmera, "gruda" ao teclado virtual e mostra na janela (e salva no vídeo, se o vídeo de saída tiver sido especificado). O operador visualiza a imagem grudada, aperta (ou não) um botão virtual, indicando a direção a seguir. O computador envia o botão virtual apertado (ou a ausência de pressionamento que estou denotando por "0") a Raspberry. Raspberry recebe o botão apertado e insere essa informação no quadro a ser enviado ao computador. A direção a seguir deve ser indicada na imagem pelo programa servidor1 antes de ser transmitida ao cliente1, para mostrar que recebeu corretamente o comando. Na figura 1, foi selecionada "ir para frente" correspondente à tecla 8.

Exercício 1 valendo 5 pontos) Mostre ao professor servidor1/cliente1 funcionando. Mostre que a gravação de vídeo funciona.

O programa abaixo é um exemplo do uso do mouse dentro do HighGUI. Há mais exemplos na apostila "highgui" e no manual do OpenCV.

```
//highgui.cpp
#include <cekeikon.h>

int estado=0; //0=nao_apertado, 1=apertou_botao_1 2=apertou_botao_2
void on_mouse(int event, int c, int l, int flags, void* userdata) {
  if (event==EVENT_LBUTTONDOWN) {
    if      (  0<=l && l<100 &&   0<=c && c<100 ) estado=1;
    else if (  0<=l && l<100 && 100<=c && c<200 ) estado=2;
    else estado=0;
  } else if (event==EVENT_LBUTTONUP) {
    estado=0;
  }
}

int main() {
  COR cinza(128,128,128);
  COR vermelho(0,0,255);
  Mat_<COR> imagem(100,200,cinza);
  namedWindow("janela",WINDOW_NORMAL);
  resizeWindow("janela",2*imagem.cols,2*imagem.rows);
  setMouseCallback("janela", on_mouse);
  imshow("janela",imagem);
  while (waitKey(1)!=27) { // ESC=27 sai do programa
    imagem.setTo(cinza);
    if (estado==1) {
      for (int l=0; l<100; l++)
```

```
        for (int c=0; c<100; c++)
          imagem(l,c)=vermelho;
    } else if (estado==2) {
      for (int l=0; l<100; l++)
        for (int c=100; c<200; c++)
          imagem(l,c)=vermelho;
    }
    imshow("janela",imagem);
  }
}
```

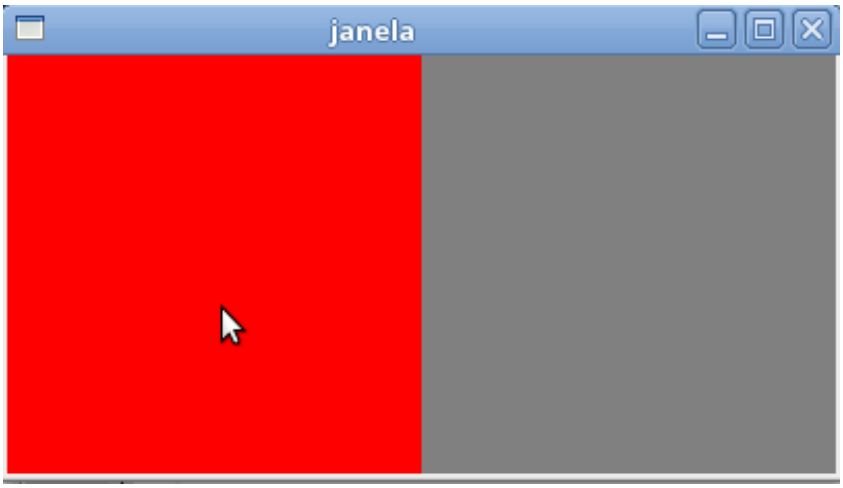

Figura 2: Janela do programa highgui.cpp que implementa dois botões controlados por mouse.

A função "on_mouse" 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.

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:

`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

4.2 Controle remoto manual do carrinho com transmissão de vídeo (fase 2)

Neste fase, vamos controlar manualmente o carrinho à distância, onde o usuário vai enxergar na tela do computador o vídeo capturado pela 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). Se necessário, modifique cliente1.cpp para cliente2.cpp. Como antes, a sintaxe para chamar os programas é:

```
raspberrypi$ servidor2
computador$ cliente2 192.168.0.110 [videosaida.avi]
```

O vídeo abaixo mostra o controle manual do carrinho:

```
http://www.lps.usp.br/hae/apostilaraspi/controlemanual3.avi
```

O funcionamento do sistema deve ser: Raspberry captura imagens da câmera, compacta e envia ao computador. Computador recebe as imagens da câmera, "gruda" ao teclado virtual e mostra na tela (e salva no vídeo, se o vídeo de saída tiver sido especificado). O operador visualiza a imagem da câmera da Raspberry, aperta (ou não) um botão virtual, indicando a direção a seguir. O computador envia a o botão virtual apertado (ou a ausência de pressionamento) a Raspberry. Raspberry recebe o botão apertado e controla os motores para executar a ação requerida.

Exercício 2 valendo 5 pontos) Mostre ao professor servidor2/cliente2 funcionando. Mostre que a gravação de vídeo funciona.