Client-Server application in C

Francesco Boi

Contents

1 Preliminaries

The project needs the installation of OpenCV4.

1.1 Installing OpenCv4 on OSX

First of all install homebrew with

```
/bin/bash -c "$(curl -fsSL https://raw.\
githubusercontent.com/Homebrew/\
install/master/install.sh)"
```

Each project requires openCV4. On OSX it can be installed using brew with the command brew install opencv4: this should already install opencv4. The compilation process uses the package pkg–config: install it with brew install pkg–config. To check whether the openCV installation was successful do:

```
pkg-config --libs --cflags opencv4
```

A long output with the folders to include and the compiled libraries is shown.

1.2 Installing OpenCV4 on Ubuntu

Currently it is not possible to install OpenCV4 through apt: one has to to download it and perform the manual installation. Detailed Instructions can be found here: Summarising, first install the dependencies

```
## Install dependencies

sudo apt -y install build-essential \
        checkinstall cmake pkg-config yasm
sudo apt -y install git gfortran
sudo apt -y install libjpeg8-dev libpng-dev

sudo apt -y install software-properties-common

sudo add-apt-repository "deb \
http://security.ubuntu.com/ubuntu xenial-security main"
sudo apt -y update

sudo apt -y install libjasper1

sudo apt -y install libtiff-dev

sudo apt -y install libavcodec-dev libavformat-dev \
```

```
        libswscale-dev libdc1394-22-dev
sudo apt -y install libxine2-dev libv4l-dev
cd /usr/include/linux
sudo ln -s -f ../libv4l1-videodev.h videodev.h
cd "$cwd"

sudo apt -y install libgstreamer1.0-dev libgstreamer\
        -plugins-base1.0-dev
sudo apt -y install libgtk2.0-dev libtbb-dev qt5-default
sudo apt -y install libatlas-base-dev
sudo apt -y install libfaac-dev \
        libmp3lame-dev libtheora-dev
sudo apt -y install libvorbis-dev \
        libxvidcore-dev
sudo apt -y install libopencore-amrnb-dev libopencore-amrwb-dev
sudo apt -y install libavresample-dev

sudo apt -y install x264 v4l-utils
# Optional dependencies
sudo apt -y install libprotobuf-dev protobuf-compiler
sudo apt -y install libgoogle-glog-dev \
        libgflags-dev
sudo apt -y install libgphoto2-dev libeigen3-dev \
        libhdf5-dev doxygen
```

Download and install OpenCV:

```
cvVersion="master"
cwd=$(pwd)
git clone https://github.com/opencv/opencv.git
cd opencv
git checkout $cvVersion
cd opencv
mkdir build
cd build
cmake -D CMAKE_BUILD_TYPE=RELEASE \
        -D CMAKE_INSTALL_PREFIX=$cwd/installation/OpenCV-"$cvVersion" \
    -D INSTALL_C_EXAMPLES=ON \
    -D INSTALL_PYTHON_EXAMPLES=ON \
```

```
    -D WITH_TBB=ON \
    -D WITH_V4L=ON \
    -D OPENCV_PYTHON3_INSTALL_PATH=$cwd/OpenCV-\
                $cvVersion-py3/lib/python3.5/site-packages \
    -D WITH_QT=ON \
    -D WITH_OPENGL=ON \
    -D OPENCV_EXTRA_MODULES_PATH=\
                ../../opencv_contrib/modules \
    -D BUILD_EXAMPLES=ON ..
make -j16
make install
\label{compilation}
```

1.3 Using older versions of OpenCV

The code itself works also with older versions of OpenCV3. If one has installed any of them and does not want to update, it is possible to use them by changing the Makefiles in the projects. Particularly in the following Makefiles:

- imgTransferC/childP/Makefile
- imgTransferC/childP/Makefile
- imgTransferC/childDB/Makefile
- imgTransferC/childDB/Makefile
- imgTransferC/imgTransferUnbuffered/Makefile (for the moment this is unused so it can be discarded).

for the lines having the command pkg–config one has to change the strings opencv4 to opencv.

1.4 Local installation of OpenCV

It is also possible to install OpeCV in a local folder. To do that, repeat the steps in **??** but change the option –DCMAKE_INSTALL_PREFIX= to another folder new folder (different from build). After that, make pkg–config commands in the Makefiles listed in 1.3 to useyourFolder/lib/pkgconfig/opencv.pc.

After that when launching the programs (see later), it might be necessary to export the dynamic libraries. This can be done on Ubunty from the Terminal with:

```
export LD_LIBRARY_PATH=/your/build/dir/openCV/openCV/build/lib/
```

On OSX substitute LD_LIBRARY_PATH with DYLD_LIBRARY_PATH.

1.5 Compilation

To compile each project it is sufficient to run make from the main folder. This will call automatically each Makefile in the subfolders. Executables are created the corresponding project folders.

2 General description

The folder contains different client-server applications. All these applications are based on the client-server application given in the book *Advanced programming in Unix Environment*. Each one has its own folder and inside each of them, there is a folder for the server application and one for the client application.

2.1 ruptime application

It is the client-server application as found in the book *Advanced programming in Unix Environment* with just few changes. When a client connects, the server launches the program in /usr/bin/uptime which returns a string made up of the current system time, the time indicating how long the system has been active, the currently active user sessions on the system and the load of the CPU. The server runs on the port 60185 and client connects to the localhost address, which is hard-coded, but it can be easily changed to connect to a remote server by changing a couple of lines in the code (this will be shown later).

After compiling, to launch the server enter the ruptime folders where the executables have been created and do ./server. The process is daemonized. To lunch the client one can use the same terminal window and do ./client userName where userName is the name of the user, i.e., the one returned by the command whoami.

The connection between the client and server is a **TCP connection**. Both server and client are written in C using the POSIX and the source codes are ruptime/dserver/server.c and ruptime/dclient/client.c.

2.2 imgConnection application

It is a client-server application that acquires images through the server webcam, transmit the data the client and the client show the stream of images in a new window. The server runs on the port 60185 and client connects to the localhost address, which is hard-coded, but it can be easily changed to connect to a remote server by changing a couple of lines in the code (this will be shown later).

After compiling, to launch the server enter the imgConnection folders where the executables have been created and do ./serverImgConnection. For debugging purposes, the demonizing feature is deactivated. To lunch the client open a new terminal window and do ./clientImgConnection.

The connection between the client and server is a TCP connection. Both server and client are written in C using the POSIX. When the client connects, the server process forks and launch the process openCVBufferedServer contained in imgTransferC/imgTransferBuffered/. This is an executable obtained from the compilation of a C++ code that exploits the OpenCV4 library to acquire the images through the webcam and the source code is the file childBufferedServer.cpp in the folder imgTransferC/imgTransferBuffered/childServer/. The image is encoded and the data is transmitted first to the parent process, by connecting the standard output of the child process to the standard input of the parent (it will be explained further later). The server application receives the data from the child process and transmit it to the client. The connection is still a **TCP connection**.

When the client receives the data, it forks and launch the process openCVBufferedClient contained in imgTransferC/imgTransferBuffered/. This is an executable obtained from the compilation of a C++ code that exploits the OpenCV4 library to show the images and the source code is the file childBufferedClient.cpp in the folder imgTransferC/imgTransferBuffered/childClient/. The image is transmitted to the child process by connecting the standard output of the parent to the standard input of the child (it will be explained further later). The child process receives the data from the child process and shows the images. The connection is still a **TCP connection**.

The child processes are called *buffered* because the writing and reading between the parent and child processes to communicate the images data are done using buffered reading and writing *POSIX* functions fwrite and fread. Alternatively, the lower level C library functions read and write can be used. For this purpose, one can use the equivalent files and processes in the folder imgTransferC/imgTransferUnbuffered. Nevertheless the also the the server and client programs should be changed to use the unbuffered lower level functions to match the ones used in their child processes.

2.3 imgConnectionless application

This application is similar to the one in 2.2. The only difference that it is a **UDP connection**, not a TCP one. The executables are in the folder imgConnectionless and the source codes are in the respective subfolders.

2.4 vid application

The application contained in this folder is actually under development . The goal is to make an application similar to 2.2 but more optimised for video streaming. Encoding and decoding each image singularly can cause too much overload. The idea is to take a short video (i.e., a more images together) and encode it, transmit it and decode it. In this way the overload causing by the encoding and decoding processes is at least reduced.

2.5 Other folders

The other folders are not client-server applications but collateral applications or libraries.

2.5.1 imgTransferC folder

This folder as already seen, contains the applications and source codes of the child processes launched by the client and server applications in 2.2, 2.3 and 2.4. Currently the latter applications use the buffered processes in the subfolder imgTransferBuffered. Alternatively, the processes in the subfolder imgTransferUnbuffered can be used by properly changing the client and server applications.

2.5.2 common folder

This folder contains common functions, constants and functionalities used by the other applications. As such it is compiled as a library and linked when the other processes are compiled.

3 POSIX networking

In this section we present the C structures and functions defined in the *POSIX* library for networking and used in the presented applications.

3.1 Socket descriptors

A socket is an abstraction of a communication endpoint. Just as they would use file descriptors to access files, applications use socket descriptors to access sockets. Socket descriptors are implemented as file descriptors in the UNIX System. Indeed, many of the functions that deal with file descriptors, such as read and write, will work with a socket descriptor. To create a socket, we call the socket function:

```
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
//Returns: file (socket) descriptor if OK, -1 on error
```

domain The `domain` argument determines the nature of the communication, including the address format. The constants start with `AF_` (for address family) because each domain has its own format for representing an address.

Domain	Description
AF_INET	IPv4 Internet domain
AF_INET6	IPv6 Internet domain (optional in POSIX.1)
AF_UNIX	UNIX domain
AF_UNSPEC	unspecified

Most systems define the `AF_LOCAL` domain also, which is an alias for `AF_UNIX` . The `AF_UNSPEC` domain is a wildcard that represents ”any” domain. Historically, some platforms provide support for additional network protocols, such as `AF_IPX` for the NetWare protocol family, but domain constants for these protocols are not defined by the POSIX.1 standard.

type The `type` argument determines the type of the socket, which further determines the communication characteristics. The socket types defined by POSIX.1 are summarized in the following table but implementations are free to add support for additional types.

Type	Description
SOCK_DGRAM	fixed-length, connectionless, unreliable messages
SOCK_RAW	datagram interface to IP (optional in POSIX.1)
SOCK_SEQPACKET	fixed-length, sequenced, reliable, connection-oriented messages
SOCK_STREAM	sequenced, reliable, bidirectional, connection-oriented byte streams

To explain their differences in more details, first consider the concept of message boundaries. A **message boundary** is the separation between two messages sent over a protocol.

With a datagram (`SOCK_DGRAM`) interface, no logical connection needs to exist between peers for them to communicate. All you need to do is send a message addressed to the socket being used by the peer process. A datagram, therefore, provides a connectionless service. Nevertheless, it has the advantage that it preserves message boundaries: if you send `FOO` and then `BAR` over SOCK_DGRAM, the other end will receive two datagrams, one containing `FOO` and the other containing `BAR`, i.e., datagrams are self-contained capsules and their boundaries are maintained, even if it is unreliable.

A byte stream (`SOCK_STREAM`), in contrast, requires that, before you can exchange data, you set up a logical connection between your socket and the socket belonging to the peer with which you wish to communicate. On the contrary, albeit the connection is reliable, no message boundary is preserved. If you send `FOO` and then `BAR` over SOCK_STREAM, the other end might get `FOO` and then `BAR`, or it might get `FOOBAR`, or `F` and then `OOB` and then `AR`. SOCK_STREAM does not make any attempt to preserve application message boundaries, it's just a stream of bytes in each direction, which means the application must manage its own boundaries on top of the stream provided[1]. As example consider the following application client-server application in Python[2]. The server code is the following:

```python
import socket
lsock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
lsock.bind(('', 9000))
lsock.listen(5)
```

[1]`https://stackoverflow.com/a/9563694/1714692`
[2]`https://stackoverflow.com/a/51662961/1714692`

```
csock, caddr = lsock.accept()
string1 = csock.recv(128)     # Receive first string
print("string1: "+str(string1))
string2 = csock.recv(128)     # Receive second string
print("string2: "+str(string2))
csock.send(b'Got your messages') # Send reply
```

and the client code is:

```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(('192.168.4.122', 9000))
s.send(b'FOO')          # Send string 1
s.send(b'BAR')          # Send string 2
reply = s.recv(128)     # Receive reply
```

The server might hang on the second `recv` call, while the client hungs on its own `recv` call. That happens because both strings the client sent (may) get bundled together and received as a single unit in the first `recv` on the server side. That is, the message boundary between the two logical messages was not preserved, and so `string1` will often contain both chunks run together: `FOOBAR`. Often there are other timing-related aspects to the code that influence when/whether that actually happens or not.

A `SOCK_SEQPACKET` socket is just like a `SOCK_STREAM` socket except that we get a message-based service instead of a byte-stream service. This means that the amount of data received from a `SOCK_SEQPACKET` socket is the same amount as was written. The Stream Control Transmission Protocol (SCTP) provides a sequential packet service in the Internet domain. SOCK_SEQPACKET is a newer technology that is not yet widely used, but tries to marry the benefits of both of the above. That is, it provides reliable, sequenced communication that also transmits entire "datagrams" as a unit (and hence maintains message boundaries).

A `SOCK_RAW` socket provides a datagram interface directly to the underlying network layer (which means IP in the Internet domain). Applications are responsible for building their own protocol headers when using this interface, because the transport protocols (TCP and UDP, for example) are bypassed. Superuser privileges are required to create a raw socket to prevent malicious applications from creating packets that might bypass established security mechanisms.

Summarising, a datagram is a self-contained message. Sending a datagram is analogous to mailing someone a letter. You can mail many letters, but you can't guarantee the order of delivery, and some might get lost along the way. Each letter contains the address of the recipient, making the letter independent from all the others. Each letter can even go to different recipients. In contrast, using a connection-oriented protocol for communicating with a peer is like making a phone call. First, you need to establish a connection by placing a phone call, but after the connection is in place, you can communicate bidirectionally with each other. The connection is a peer-to-peer communication channel over which you talk. Your words contain no addressing information, as a point-to-point virtual connection exists between both ends of the call, and the connection itself implies a particular source and destination. A SOCK_STREAM socket provides a byte-stream service; applications are unaware of message boundaries. This means that when we read data from a SOCK_STREAM socket, it might not return the same number of bytes written by the sender. We will eventually get everything sent to us, but it might take several function calls. A SOCK_SEQPACKET is a way in between the two.

protocol The `protocol` argument is usually zero, to select the default protocol for the given domain and socket type. When multiple protocols are supported for the same domain and socket type, we can use the protocol argument to select a particular protocol. The default protocol for a `SOCK_STREAM` socket in the `AF_INET` communication domain is TCP (Transmission Control Protocol). The default protocol for a `SOCK_DGRAM` socket in the `AF_INET` communication domain is UDP (User Datagram Protocol). Table 1 lists the protocols defined for the Internet domain sockets.

Protocol	**Description**
IPPROTO_IP	IPv4 Internet Protocol
IPPROTO_IPV6	IPv6 Internet Protocol (optional in POSIX.1)
IPPROTO_ICMP	Internet Control Message Protocol
IPPROTO_RAW	Raw IP packets protocol (optional in POSIX.1)
IPPROTO_TCP	Transmission Control Protocol
IPPROTO_UDP	User Datagram Protocol

Table 1

Calling socket is similar to calling `open`. In both cases, you get a file descriptor that can be used for I/O. When you are done using the file descriptor, you call close to relinquish access to the file or socket and free up the file descriptor for reuse. Communication on a socket is bidirectional. We can disable I/O on a socket with the `shutdow` function.

```
#include <sys/socket.h>
int shutdown(int sockfd, int how);
//Returns: 0 if OK, -1 on error
```

how If `how` is `SHUT_RD`, then reading from the socket is disabled. If how is `SHUT_WR`, then we can't use the socket for transmitting data. We can use `SHUT_RDWR` to disable both data transmission and reception.

`close` will deallocate the network endpoint only when the last active reference is closed. If we duplicate the socket (with `dup`, for example), the socket won't be deallocated until we close the last file descriptor referring to it. The `shutdown` function allows us to deactivate a socket independently of the number of active file descriptors referencing it.

3.2 Addressing

In the previous section, we learned how to create and destroy a socket. Before we learn to do something useful with a socket, we need to learn how to identify the process with which we wish to communicate. Identifying the process has two components. The machine's network address helps us identify the computer on the network we wish to contact, and the service, represented by a port number, helps us identify the particular process on the computer.

An address identifies a socket endpoint in a particular communication domain. The address format is specific to the particular domain. So that addresses with different formats can be passed to the socket functions, the addresses are cast to a generic `sockaddr` address structure:

```
struct sockaddr {
sa_family_t sa_family; /* address family */
char sa_data[]; /* variable-length address */
```

```
        (*:*)
        };
```

Implementations are free to add more members and define a size for the `sa_data` member. For example, on Linux, the structure is defined as:

```
    struct sockaddr {
      sa_family_t  sa_family;    /* address family */
```

```
    char           sa_data[14];  /* variable-length address
        */
};
```

Internet addresses are defined in `<netinet/in.h>`. In the IPv4 Internet domain (`AF_INET`), a socket address is represented by a `sockaddr_in` structure:

```
struct in_addr {
  in_addr_t        s_addr;         /* IPv4 address */
};

struct sockaddr_in {
  sa_family_t     sin_family;     /* address family */
  in_port_t       sin_port;       /* port number */
  struct in_addr  sin_addr;       /* IPv4 address */
};
```

The `in_port_t` data type is defined to be a `uint16_t`. The `in_addr_t` data type is defined to be a `uint32_t`. These integer data types specify the number of bits in the data type and are defined in `<stdint.h>`. In contrast to the `AF_INET` domain, the IPv6 Internet domain (`AF_INET6`) socket address is represented by a `sockaddr_in6` structure:

```
struct in6_addr {
  uint8_t        s6_addr[16];     /* IPv46address */
};

struct sockaddr_in6 {
  sa_family_t     sin6_family;    /* address family */
  in_port_t       sin6_port;      /* port number */
  uint32_t       sin6_flowinfo;  /*traffic class and flow
      info*/
  struct in6_addr sin6_addr;     /* IPv6 address */
  uint32_t       sin6_scope_id;  /* set of interfaces for
      scope*/
};
```

Again, individual implementations are free to add more fields.

Note that although the `sockaddr_in` and `sockaddr_in6` structures are quite different, they are both passed to the socket routines cast to a `sockaddr` structure.

3.3 Address lookup

Ideally, an application won't have to be aware of the internal structure of a socket address, so that it will work with many different protocols that provide the same type of service. Network configurations can be retrieved from static files such as `/etc/services`, that provides a mapping between human-friendly textual names for internet services, and their underlying assigned port numbers and protocol types, and `/etc/hosts` that maps hostnames to IP addresses. The latter is one of several system facilities that assists in addressing network nodes in a computer network. Alternatively its function can be manages by a name service, such as DNS (Domain Name Service). The hosts known by a by a computer system are found by calling `gethostent`.

```
#include <netdb.h>
struct hostent *gethostent(void);
//Returns: pointer if OK, NULL on error

void sethostent(int stayopen);
void endhostent(void);
```

If the host database file isn't already open, `gethostent`will open it. The `gethostent` function returns the next entry in the file. The `sethostent` function will open the file or rewind it if it is already open. When the stayopen argument is set to a nonzero value, the file remains open after calling `gethostent`. The `endhostent` function can be used to close the file. When `gethostent` returns, we get a pointer to a hostent structure, which might point to a static data buffer that is overwritten each time we call `gethostent`. The hostent structure is defined to have at least the following members:

```
struct hostent {
  char   *h_name;       /* name of host */
  char  **h_aliases;    /* pointer to alternate host name
      array */
  int     h_addrtype;   /* address type */
  int     h_length;     /* length in bytes of address */
  char  **h_addr_list;  /* pointer to array of network
      addresses */
  ⋮
};
```

The addresses returned are in network byte order. `gethostbyname` and `gethostbyaddr` are now considered to be obsolete. We can get network names and numbers with a similar set of interfaces.

```c
#include <netdb.h>
struct netent *getnetbyaddr(uint32_t net, int type);
struct netent *getnetbyname(const char *name);
struct netent *getnetent(void);
//All return: pointer if OK, NULL on error

void setnetent(int stayopen);
void endnetent(void);
```

The netent structure contains at least the following fields:

```c
struct netent {
        char *n_name;   /* network name */
        char **n_aliases;       /* alternate network name
           array pointer */
        int n_addrtype; /* address type */
        uint32_t  n_net;        /* network number */
        ⋮
};
```

The network number is returned in network byte order.

Services are represented by the port number portion of the address. Each service is offered on a unique, well-known port number. We can map a service name to a port number with `getservbyname`, map a port number to a service name with `getservbyport`, or scan the services database sequentially with `getservent`.

```c
#include <netdb.h>
struct servent *getservbyname(const char *name, const char
   *proto);
struct servent *getservbyport(int port, const char *proto);
struct servent *getservent(void);
void setservent(int stayopen); void endservent(void);
//All return: pointer if OK, NULL on error

void setservent(int stayopen);
void endservent(void);
```

The `servent` structure is defined to have at least the following members:

```c
struct servent {
     char   *s_name;     /* service name */
     char  **s_aliases;  /* pointer to alternate service
        name array */
     int     s_port;     /* port number */
     char   *s_proto;    /* name of protocol */
```

```
(*⋮*)
};
```

POSIX.1 defines several new functions to allow an application to map from a host name and a service name to an address, and vice versa. These functions replace the older `gethostbyname` and `gethostbyaddr` functions. The `getaddrinfo` function allows us to map a host name and a service name to an address.

```
#include <sys/socket.h>
#include <netdb.h>
int getaddrinfo(const char *restrict host,
        const char *restrict service,
        const struct addrinfo *restrict hint,
        struct addrinfo **restrict res);
//Returns: 0 if OK, nonzero error code on error

void freeaddrinfo(struct addrinfo *ai);
```

We need to provide the host name, the service name, or both. If we provide only one name, the other should be a null pointer. The host name can be either a node name or the host address in dotted-decimal notation. The `getaddrinfo` function returns a linked list of `addrinfo` structures. We can use `freeaddrinfo` to free one or more of these structures, depending on how many structures are linked together using the `ai_next` field in the structures.

The `addrinfo` structure is defined to include at least the following members:

```
struct addrinfo {
    int          ai_flags;
    int          ai_family;
    int          ai_socktype;
    int          ai_protocol;
    socklen_t    ai_addrlen;
    struct sockaddr *ai_addr;
    char *ai_canonname;  /* canonical name of host */
    struct addrinfo *ai_next;  /* next in list */
        ⋮
};
```

We can supply an optional hint to select addresses that meet certain criteria. The hint is a template used for filtering addresses and uses only the `ai_family`, `ai_flags`, `ai_protocol`, and `ai_socktype` fields. The remaining integer fields must be set to 0, and the pointer fields must be null. Table 2 summarizes the flags we can use in the `ai_flags` field to customize how addresses and names are treated.

Flag	Description
AI_ADDRCONFIG	Query for whichever address type (IPv4 or IPv6) is configured.
AI_ALL	Look for both IPv4 and IPv6 addresses (used only with AI_V4MAPPED).
AI_CANONNAME	Request a canonical name (as opposed to an alias).
AI_NUMERICHOST	The host address is specified in numeric format; don't try to translate it.
AI_NUMERICSERV	The service is specified as a numeric port number; don't try to translate it.
AI_PASSIVE	Socket address is intended to be bound for listening.
AI_V4MAPPED	If no IPv6 addresses are found, return IPv4 addresses mapped in IPv6 format.

Table 2

If `getaddrinfo` fails, we need to call `gai_strerror` to convert the error code returned into an error. message.

```
#include <netdb.h>
const char *gai_strerror(int error);
//Returns: a pointer to a string describing the error
```

The `getnameinfo` function converts an address into host and service names.

```
#include <sys/socket.h>
#include <netdb.h>
int getnameinfo(const struct sockaddr *restrict addr,
        socklen_t alen, char *restrict host,
        socklen_t hostlen, char *restrict service,
        socklen_t servlen, int flags);
//Returns: 0 if OK, nonzero on error
```

The socket address (`addr`) is translated into a host name and a service name. If `host` is non-null, it points to a buffer `hostlen` bytes long that will be used to return the host name. Similarly, if `service` is non-null, it points to a buffer `servlen` bytes long that will be used to return the service name. The `flags` argument gives us some control over how the translation is done: they work as a mask, so that we . Table 3 summarizes the supported flags.

Flag	Description
NI_DGRAM	The service is datagram based instead of stream based.
NI_NAMEREQD	If the host name can't be found, treat this as an error.
NI_NOFQDN	Return only the node name portion of the fully qualified domain name for local hosts.
NI_NUMERICHOST	Return the numeric form of the host address instead of the name.
NI_NUMERICSCOPE	For IPv6, return the numeric form of the scope ID instead of the name.
NI_NUMERICSERV	Return the numeric form of the service address (i.e., the port number) instead of the name.

Table 3

4 Analysis of the imgConnection application

In this section the code building up the `imcgConnection` application is explained in details. Much of it is common to all the applications.