#define \_XOPEN\_SOURCE

#include "image.h"

#include "http\_stream.h"

//

// a single-threaded, multi client(using select), debug webserver - streaming out mjpg.

// on win, \_WIN32 has to be defined, must link against ws2\_32.lib (socks on linux are for free)

//

#include <cstdio>

#include <vector>

#include <iostream>

#include <algorithm>

#include <memory>

#include <mutex>

#include <thread>

#include <atomic>

#include <ctime>

using std::cerr;

using std::endl;

//

// socket related abstractions:

//

#ifdef \_WIN32

#ifndef USE\_CMAKE\_LIBS

#pragma comment(lib, "ws2\_32.lib")

#endif

#define WIN32\_LEAN\_AND\_MEAN

#define \_WINSOCK\_DEPRECATED\_NO\_WARNINGS

#include <windows.h>

#include <winsock2.h>

#include <ws2tcpip.h>

#include "gettimeofday.h"

#define PORT unsigned long

#define ADDRPOINTER int\*

struct \_INIT\_W32DATA

{

WSADATA w;

\_INIT\_W32DATA() { WSAStartup(MAKEWORD(2, 1), &w); }

} \_init\_once;

// Graceful closes will first close their output channels and then wait for the peer

// on the other side of the connection to close its output channels. When both sides are done telling

// each other they won,t be sending any more data (i.e., closing output channels),

// the connection can be closed fully, with no risk of reset.

static int close\_socket(SOCKET s) {

int close\_output = ::shutdown(s, 1); // 0 close input, 1 close output, 2 close both

char \*buf = (char \*)calloc(1024, sizeof(char));

::recv(s, buf, 1024, 0);

free(buf);

int close\_input = ::shutdown(s, 0);

int result = ::closesocket(s);

cerr << "Close socket: out = " << close\_output << ", in = " << close\_input << " \n";

return result;

}

#else // \_WIN32 - else: nix

#include "darkunistd.h"

#include <fcntl.h>

#include <sys/time.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netdb.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <signal.h>

#define PORT unsigned short

#define SOCKET int

#define HOSTENT struct hostent

#define SOCKADDR struct sockaddr

#define SOCKADDR\_IN struct sockaddr\_in

#define ADDRPOINTER unsigned int\*

#define INVALID\_SOCKET -1

#define SOCKET\_ERROR -1

struct \_IGNORE\_PIPE\_SIGNAL

{

struct sigaction new\_actn, old\_actn;

\_IGNORE\_PIPE\_SIGNAL() {

new\_actn.sa\_handler = SIG\_IGN; // ignore the broken pipe signal

sigemptyset(&new\_actn.sa\_mask);

new\_actn.sa\_flags = 0;

sigaction(SIGPIPE, &new\_actn, &old\_actn);

// sigaction (SIGPIPE, &old\_actn, NULL); // - to restore the previous signal handling

}

} \_init\_once;

static int close\_socket(SOCKET s) {

int close\_output = ::shutdown(s, 1); // 0 close input, 1 close output, 2 close both

char \*buf = (char \*)calloc(1024, sizeof(char));

::recv(s, buf, 1024, 0);

free(buf);

int close\_input = ::shutdown(s, 0);

int result = close(s);

std::cerr << "Close socket: out = " << close\_output << ", in = " << close\_input << " \n";

return result;

}

#endif // \_WIN32

class JSON\_sender

{

SOCKET sock;

SOCKET maxfd;

fd\_set master;

int timeout; // master sock timeout, shutdown after timeout usec.

int close\_all\_sockets;

int \_write(int sock, char const\*const s, int len)

{

if (len < 1) { len = strlen(s); }

return ::send(sock, s, len, 0);

}

public:

JSON\_sender(int port = 0, int \_timeout = 400000)

: sock(INVALID\_SOCKET)

, timeout(\_timeout)

{

close\_all\_sockets = 0;

FD\_ZERO(&master);

if (port)

open(port);

}

~JSON\_sender()

{

close\_all();

release();

}

bool release()

{

if (sock != INVALID\_SOCKET)

::shutdown(sock, 2);

sock = (INVALID\_SOCKET);

return false;

}

void close\_all()

{

close\_all\_sockets = 1;

write("\n]"); // close JSON array

}

bool open(int port)

{

sock = ::socket(AF\_INET, SOCK\_STREAM, IPPROTO\_TCP);

SOCKADDR\_IN address;

address.sin\_addr.s\_addr = INADDR\_ANY;

address.sin\_family = AF\_INET;

address.sin\_port = htons(port); // ::htons(port);

int reuse = 1;

if (setsockopt(sock, SOL\_SOCKET, SO\_REUSEADDR, (const char\*)&reuse, sizeof(reuse)) < 0)

cerr << "setsockopt(SO\_REUSEADDR) failed" << endl;

// Non-blocking sockets

// Windows: ioctlsocket() and FIONBIO

// Linux: fcntl() and O\_NONBLOCK

#ifdef WIN32

unsigned long i\_mode = 1;

int result = ioctlsocket(sock, FIONBIO, &i\_mode);

if (result != NO\_ERROR) {

std::cerr << "ioctlsocket(FIONBIO) failed with error: " << result << std::endl;

}

#else // WIN32

int flags = fcntl(sock, F\_GETFL, 0);

fcntl(sock, F\_SETFL, flags | O\_NONBLOCK);

#endif // WIN32

#ifdef SO\_REUSEPORT

if (setsockopt(sock, SOL\_SOCKET, SO\_REUSEPORT, (const char\*)&reuse, sizeof(reuse)) < 0)

cerr << "setsockopt(SO\_REUSEPORT) failed" << endl;

#endif

if (::bind(sock, (SOCKADDR\*)&address, sizeof(SOCKADDR\_IN)) == SOCKET\_ERROR)

{

cerr << "error JSON\_sender: couldn't bind sock " << sock << " to port " << port << "!" << endl;

return release();

}

if (::listen(sock, 10) == SOCKET\_ERROR)

{

cerr << "error JSON\_sender: couldn't listen on sock " << sock << " on port " << port << " !" << endl;

return release();

}

FD\_ZERO(&master);

FD\_SET(sock, &master);

maxfd = sock;

return true;

}

bool isOpened()

{

return sock != INVALID\_SOCKET;

}

bool write(char const\* outputbuf)

{

fd\_set rread = master;

struct timeval select\_timeout = { 0, 0 };

struct timeval socket\_timeout = { 0, timeout };

if (::select(maxfd + 1, &rread, NULL, NULL, &select\_timeout) <= 0)

return true; // nothing broken, there's just noone listening

int outlen = static\_cast<int>(strlen(outputbuf));

#ifdef \_WIN32

for (unsigned i = 0; i<rread.fd\_count; i++)

{

int addrlen = sizeof(SOCKADDR);

SOCKET s = rread.fd\_array[i]; // fd\_set on win is an array, while ...

#else

for (int s = 0; s <= maxfd; s++)

{

socklen\_t addrlen = sizeof(SOCKADDR);

if (!FD\_ISSET(s, &rread)) // ... on linux it's a bitmask ;)

continue;

#endif

if (s == sock) // request on master socket, accept and send main header.

{

SOCKADDR\_IN address = { 0 };

SOCKET client = ::accept(sock, (SOCKADDR\*)&address, &addrlen);

if (client == SOCKET\_ERROR)

{

cerr << "error JSON\_sender: couldn't accept connection on sock " << sock << " !" << endl;

return false;

}

if (setsockopt(client, SOL\_SOCKET, SO\_RCVTIMEO, (char \*)&socket\_timeout, sizeof(socket\_timeout)) < 0) {

cerr << "error JSON\_sender: SO\_RCVTIMEO setsockopt failed\n";

}

if (setsockopt(client, SOL\_SOCKET, SO\_SNDTIMEO, (char \*)&socket\_timeout, sizeof(socket\_timeout)) < 0) {

cerr << "error JSON\_sender: SO\_SNDTIMEO setsockopt failed\n";

}

maxfd = (maxfd>client ? maxfd : client);

FD\_SET(client, &master);

\_write(client, "HTTP/1.0 200 OK\r\n", 0);

\_write(client,

"Server: Mozarella/2.2\r\n"

"Accept-Range: bytes\r\n"

"Connection: close\r\n"

"Max-Age: 0\r\n"

"Expires: 0\r\n"

"Cache-Control: no-cache, private\r\n"

"Pragma: no-cache\r\n"

"Content-Type: application/json\r\n"

//"Content-Type: multipart/x-mixed-replace; boundary=boundary\r\n"

"\r\n", 0);

\_write(client, "[\n", 0); // open JSON array

int n = \_write(client, outputbuf, outlen);

cerr << "JSON\_sender: new client " << client << endl;

}

else // existing client, just stream pix

{

//char head[400];

// application/x-resource+json or application/x-collection+json - when you are representing REST resources and collections

// application/json or text/json or text/javascript or text/plain.

// https://stackoverflow.com/questions/477816/what-is-the-correct-json-content-type

//sprintf(head, "\r\nContent-Length: %zu\r\n\r\n", outlen);

//sprintf(head, "--boundary\r\nContent-Type: application/json\r\nContent-Length: %zu\r\n\r\n", outlen);

//\_write(s, head, 0);

if (!close\_all\_sockets) \_write(s, ", \n", 0);

int n = \_write(s, outputbuf, outlen);

if (n < (int)outlen)

{

cerr << "JSON\_sender: kill client " << s << endl;

close\_socket(s);

//::shutdown(s, 2);

FD\_CLR(s, &master);

}

if (close\_all\_sockets) {

int result = close\_socket(s);

cerr << "JSON\_sender: close clinet: " << result << " \n";

continue;

}

}

}

if (close\_all\_sockets) {

int result = close\_socket(sock);

cerr << "JSON\_sender: close acceptor: " << result << " \n\n";

}

return true;

}

};

// ----------------------------------------

static std::unique\_ptr<JSON\_sender> js\_ptr;

static std::mutex mtx;

void delete\_json\_sender()

{

std::lock\_guard<std::mutex> lock(mtx);

js\_ptr.release();

}

void send\_json\_custom(char const\* send\_buf, int port, int timeout)

{

try {

std::lock\_guard<std::mutex> lock(mtx);

if(!js\_ptr) js\_ptr.reset(new JSON\_sender(port, timeout));

js\_ptr->write(send\_buf);

}

catch (...) {

cerr << " Error in send\_json\_custom() function \n";

}

}

void send\_json(detection \*dets, int nboxes, int classes, char \*\*names, long long int frame\_id, int port, int timeout)

{

try {

char \*send\_buf = detection\_to\_json(dets, nboxes, classes, names, frame\_id, NULL);

send\_json\_custom(send\_buf, port, timeout);

std::cout << " JSON-stream sent. \n";

free(send\_buf);

}

catch (...) {

cerr << " Error in send\_json() function \n";

}

}

// ----------------------------------------

#ifdef OPENCV

#include <opencv2/opencv.hpp>

#include <opencv2/highgui/highgui.hpp>

#include <opencv2/highgui/highgui\_c.h>

#include <opencv2/imgproc/imgproc\_c.h>

#ifndef CV\_VERSION\_EPOCH

#include <opencv2/videoio/videoio.hpp>

#endif

using namespace cv;

class MJPG\_sender

{

SOCKET sock;

SOCKET maxfd;

fd\_set master;

int timeout; // master sock timeout, shutdown after timeout usec.

int quality; // jpeg compression [1..100]

int close\_all\_sockets;

int \_write(int sock, char const\*const s, int len)

{

if (len < 1) { len = strlen(s); }

return ::send(sock, s, len, 0);

}

public:

MJPG\_sender(int port = 0, int \_timeout = 400000, int \_quality = 30)

: sock(INVALID\_SOCKET)

, timeout(\_timeout)

, quality(\_quality)

{

close\_all\_sockets = 0;

FD\_ZERO(&master);

if (port)

open(port);

}

~MJPG\_sender()

{

close\_all();

release();

}

bool release()

{

if (sock != INVALID\_SOCKET)

::shutdown(sock, 2);

sock = (INVALID\_SOCKET);

return false;

}

void close\_all()

{

close\_all\_sockets = 1;

cv::Mat tmp(cv::Size(10, 10), CV\_8UC3);

write(tmp);

}

bool open(int port)

{

sock = ::socket(AF\_INET, SOCK\_STREAM, IPPROTO\_TCP);

SOCKADDR\_IN address;

address.sin\_addr.s\_addr = INADDR\_ANY;

address.sin\_family = AF\_INET;

address.sin\_port = htons(port); // ::htons(port);

int reuse = 1;

if (setsockopt(sock, SOL\_SOCKET, SO\_REUSEADDR, (const char\*)&reuse, sizeof(reuse)) < 0)

cerr << "setsockopt(SO\_REUSEADDR) failed" << endl;

// Non-blocking sockets

// Windows: ioctlsocket() and FIONBIO

// Linux: fcntl() and O\_NONBLOCK

#ifdef WIN32

unsigned long i\_mode = 1;

int result = ioctlsocket(sock, FIONBIO, &i\_mode);

if (result != NO\_ERROR) {

std::cerr << "ioctlsocket(FIONBIO) failed with error: " << result << std::endl;

}

#else // WIN32

int flags = fcntl(sock, F\_GETFL, 0);

fcntl(sock, F\_SETFL, flags | O\_NONBLOCK);

#endif // WIN32

#ifdef SO\_REUSEPORT

if (setsockopt(sock, SOL\_SOCKET, SO\_REUSEPORT, (const char\*)&reuse, sizeof(reuse)) < 0)

cerr << "setsockopt(SO\_REUSEPORT) failed" << endl;

#endif

if (::bind(sock, (SOCKADDR\*)&address, sizeof(SOCKADDR\_IN)) == SOCKET\_ERROR)

{

cerr << "error MJPG\_sender: couldn't bind sock " << sock << " to port " << port << "!" << endl;

return release();

}

if (::listen(sock, 10) == SOCKET\_ERROR)

{

cerr << "error MJPG\_sender: couldn't listen on sock " << sock << " on port " << port << " !" << endl;

return release();

}

FD\_ZERO(&master);

FD\_SET(sock, &master);

maxfd = sock;

return true;

}

bool isOpened()

{

return sock != INVALID\_SOCKET;

}

bool write(const Mat & frame)

{

fd\_set rread = master;

struct timeval select\_timeout = { 0, 0 };

struct timeval socket\_timeout = { 0, timeout };

if (::select(maxfd + 1, &rread, NULL, NULL, &select\_timeout) <= 0)

return true; // nothing broken, there's just noone listening

std::vector<uchar> outbuf;

std::vector<int> params;

params.push\_back(IMWRITE\_JPEG\_QUALITY);

params.push\_back(quality);

cv::imencode(".jpg", frame, outbuf, params); //REMOVED FOR COMPATIBILITY

// https://docs.opencv.org/3.4/d4/da8/group\_\_imgcodecs.html#ga292d81be8d76901bff7988d18d2b42ac

//std::cerr << "cv::imencode call disabled!" << std::endl;

int outlen = static\_cast<int>(outbuf.size());

#ifdef \_WIN32

for (unsigned i = 0; i<rread.fd\_count; i++)

{

int addrlen = sizeof(SOCKADDR);

SOCKET s = rread.fd\_array[i]; // fd\_set on win is an array, while ...

#else

for (int s = 0; s <= maxfd; s++)

{

socklen\_t addrlen = sizeof(SOCKADDR);

if (!FD\_ISSET(s, &rread)) // ... on linux it's a bitmask ;)

continue;

#endif

if (s == sock) // request on master socket, accept and send main header.

{

SOCKADDR\_IN address = { 0 };

SOCKET client = ::accept(sock, (SOCKADDR\*)&address, &addrlen);

if (client == SOCKET\_ERROR)

{

cerr << "error MJPG\_sender: couldn't accept connection on sock " << sock << " !" << endl;

return false;

}

if (setsockopt(client, SOL\_SOCKET, SO\_RCVTIMEO, (char \*)&socket\_timeout, sizeof(socket\_timeout)) < 0) {

cerr << "error MJPG\_sender: SO\_RCVTIMEO setsockopt failed\n";

}

if (setsockopt(client, SOL\_SOCKET, SO\_SNDTIMEO, (char \*)&socket\_timeout, sizeof(socket\_timeout)) < 0) {

cerr << "error MJPG\_sender: SO\_SNDTIMEO setsockopt failed\n";

}

maxfd = (maxfd>client ? maxfd : client);

FD\_SET(client, &master);

\_write(client, "HTTP/1.0 200 OK\r\n", 0);

\_write(client,

"Server: Mozarella/2.2\r\n"

"Accept-Range: bytes\r\n"

"Connection: close\r\n"

"Max-Age: 0\r\n"

"Expires: 0\r\n"

"Cache-Control: no-cache, private\r\n"

"Pragma: no-cache\r\n"

"Content-Type: multipart/x-mixed-replace; boundary=mjpegstream\r\n"

"\r\n", 0);

cerr << "MJPG\_sender: new client " << client << endl;

}

else // existing client, just stream pix

{

if (close\_all\_sockets) {

int result = close\_socket(s);

cerr << "MJPG\_sender: close clinet: " << result << " \n";

continue;

}

char head[400];

sprintf(head, "--mjpegstream\r\nContent-Type: image/jpeg\r\nContent-Length: %zu\r\n\r\n", outlen);

\_write(s, head, 0);

int n = \_write(s, (char\*)(&outbuf[0]), outlen);

cerr << "known client: " << s << ", sent = " << n << ", must be sent outlen = " << outlen << endl;

if (n < (int)outlen)

{

cerr << "MJPG\_sender: kill client " << s << endl;

//::shutdown(s, 2);

close\_socket(s);

FD\_CLR(s, &master);

}

}

}

if (close\_all\_sockets) {

int result = close\_socket(sock);

cerr << "MJPG\_sender: close acceptor: " << result << " \n\n";

}

return true;

}

};

// ----------------------------------------

static std::mutex mtx\_mjpeg;

//struct mat\_cv : cv::Mat { int a[0]; };

void send\_mjpeg(mat\_cv\* mat, int port, int timeout, int quality)

{

try {

std::lock\_guard<std::mutex> lock(mtx\_mjpeg);

static MJPG\_sender wri(port, timeout, quality);

//cv::Mat mat = cv::cvarrToMat(ipl);

wri.write(\*(cv::Mat\*)mat);

std::cout << " MJPEG-stream sent. \n";

}

catch (...) {

cerr << " Error in send\_mjpeg() function \n";

}

}

// ----------------------------------------

std::string get\_system\_frame\_time\_string()

{

std::time\_t t = std::chrono::system\_clock::to\_time\_t(std::chrono::system\_clock::now());

static std::mutex mtx;

std::lock\_guard<std::mutex> lock(mtx);

struct tm \*tmp\_buf = localtime(&t);

char buff[256];

std::strftime(buff, 256, "%A %F %T", tmp\_buf);

std::string system\_frame\_time = buff;

return system\_frame\_time;

}

// ----------------------------------------

#ifdef \_\_CYGWIN\_\_

int send\_http\_post\_request(char \*http\_post\_host, int server\_port, const char \*videosource,

detection \*dets, int nboxes, int classes, char \*\*names, long long int frame\_id, int ext\_output, int timeout)

{

std::cerr << " send\_http\_post\_request() isn't implemented \n";

return 0;

}

#else // \_\_CYGWIN\_\_

#ifndef NI\_MAXHOST

#define NI\_MAXHOST 1025

#endif

#ifndef NI\_NUMERICHOST

#define NI\_NUMERICHOST 0x02

#endif

//#define CPPHTTPLIB\_OPENSSL\_SUPPORT

#include "httplib.h"

// https://webhook.site/

// https://github.com/yhirose/cpp-httplib

// sent POST http request

int send\_http\_post\_request(char \*http\_post\_host, int server\_port, const char \*videosource,

detection \*dets, int nboxes, int classes, char \*\*names, long long int frame\_id, int ext\_output, int timeout)

{

const float thresh = 0.005; // function get\_network\_boxes() has already filtred dets by actual threshold

std::string message;

for (int i = 0; i < nboxes; ++i) {

char labelstr[4096] = { 0 };

int class\_id = -1;

for (int j = 0; j < classes; ++j) {

int show = strncmp(names[j], "dont\_show", 9);

if (dets[i].prob[j] > thresh && show) {

if (class\_id < 0) {

strcat(labelstr, names[j]);

class\_id = j;

char buff[10];

sprintf(buff, " (%2.0f%%)", dets[i].prob[j] \* 100);

strcat(labelstr, buff);

}

else {

strcat(labelstr, ", ");

strcat(labelstr, names[j]);

}

printf("%s: %.0f%% ", names[j], dets[i].prob[j] \* 100);

}

}

if (class\_id >= 0) {

message += std::string(names[class\_id]) + std::string(", id: ") + std::to\_string(class\_id) + "\n";

}

}

if (!message.empty())

{

std::string time = get\_system\_frame\_time\_string();

message += "\ntime:\n" + time + "\n";

message += "videosource:\n" + std::string(videosource);

std::string http\_post\_host\_str = http\_post\_host;

int slash\_index = http\_post\_host\_str.find("/");

std::string http\_path = http\_post\_host\_str.substr(slash\_index, http\_post\_host\_str.length() - slash\_index);

http\_post\_host\_str = http\_post\_host\_str.substr(0, slash\_index);

// send HTTP-Post request

httplib::Client cli(http\_post\_host\_str.c\_str(), server\_port, timeout);

auto res = cli.Post(http\_path.c\_str(), message, "text/plain");

return 1;

}

return 0;

}

#endif // \_\_CYGWIN\_\_

#endif // OPENCV

// -----------------------------------------------------

#if \_\_cplusplus >= 201103L || \_MSC\_VER >= 1900 // C++11

#include <chrono>

#include <iostream>

static std::chrono::steady\_clock::time\_point steady\_start, steady\_end;

static double total\_time;

double get\_time\_point() {

std::chrono::steady\_clock::time\_point current\_time = std::chrono::steady\_clock::now();

//uint64\_t now = std::chrono::duration\_cast<std::chrono::milliseconds>(current\_time.time\_since\_epoch()).count();

return std::chrono::duration\_cast<std::chrono::microseconds>(current\_time.time\_since\_epoch()).count();

}

void start\_timer() {

steady\_start = std::chrono::steady\_clock::now();

}

void stop\_timer() {

steady\_end = std::chrono::steady\_clock::now();

}

double get\_time() {

double took\_time = std::chrono::duration<double>(steady\_end - steady\_start).count();

total\_time += took\_time;

return took\_time;

}

void stop\_timer\_and\_show() {

stop\_timer();

std::cout << " " << get\_time() \* 1000 << " msec" << std::endl;

}

void stop\_timer\_and\_show\_name(char \*name) {

stop\_timer();

std::cout << " " << name;

std::cout << " " << get\_time() \* 1000 << " msec" << std::endl;

}

void show\_total\_time() {

std::cout << " Total: " << total\_time \* 1000 << " msec" << std::endl;

}

int custom\_create\_thread(custom\_thread\_t \* tid, const custom\_attr\_t \* attr, void \*(\*func) (void \*), void \*arg)

{

std::thread \*ptr = new std::thread(func, arg);

\*tid = (custom\_thread\_t \*)ptr;

if (tid) return 0;

else return -1;

}

int custom\_join(custom\_thread\_t tid, void \*\*value\_ptr)

{

std::thread \*ptr = (std::thread \*)tid;

if (ptr) {

ptr->join();

delete ptr;

return 0;

}

else printf(" Error: ptr of thread is NULL in custom\_join() \n");

return -1;

}

int custom\_atomic\_load\_int(volatile int\* obj)

{

const volatile std::atomic<int>\* ptr\_a = (const volatile std::atomic<int>\*)obj;

return std::atomic\_load(ptr\_a);

}

void custom\_atomic\_store\_int(volatile int\* obj, int desr)

{

volatile std::atomic<int>\* ptr\_a = (volatile std::atomic<int>\*)obj;

std::atomic\_store(ptr\_a, desr);

}

void this\_thread\_sleep\_for(int ms\_time)

{

std::chrono::milliseconds dura(ms\_time);

std::this\_thread::sleep\_for(dura);

}

void this\_thread\_yield()

{

std::this\_thread::yield();

}

#else // C++11

#include <iostream>

double get\_time\_point() { return 0; }

void start\_timer() {}

void stop\_timer() {}

double get\_time() { return 0; }

void stop\_timer\_and\_show() {

std::cout << " stop\_timer\_and\_show() isn't implemented " << std::endl;

}

void stop\_timer\_and\_show\_name(char \*name) { stop\_timer\_and\_show(); }

void total\_time() {}

#endif // C++11