#ifndef \_\_SOCKET\_SERVER\_H\_\_

#define \_\_SOCKET\_SERVER\_H\_\_

#include "SocketBase.h"

struct RecvData

{

HSocket socketClient;

int dataLen;

char data[1024];

};

class SocketServer : public SocketBase

{

public:

static SocketServer\* getInstance();

void destroyInstance();

bool startServer(unsigned short port);

void sendMessage(HSocket socket, const char\* data, int count);

void sendMessage(const char\* data, int count);

void update(float dt);

std::function<void(const char\* ip)> onStart;

std::function<void(HSocket socket)> onNewConnection;

std::function<void(HSocket socket, const char\* data, int count)> onRecv;

std::function<void(HSocket socket)> onDisconnect;

CC\_CONSTRUCTOR\_ACCESS:

SocketServer();

~SocketServer();

private:

void clear();

bool initServer(unsigned short port);

void acceptClient();

void acceptFunc();

void newClientConnected(HSocket socket);

void recvMessage(HSocket socket);

private:

static SocketServer\* s\_server;

HSocket \_socketServer;

unsigned short \_serverPort;

private:

std::list<HSocket> \_clientSockets;

std::list<SocketMessage\*> \_UIMessageQueue;

std::mutex \_UIMessageQueueMutex;

};

#endif

#include "SocketServer.h"

SocketServer\* SocketServer::s\_server = nullptr;

SocketServer\* SocketServer::getInstance()

{

if (s\_server == nullptr)

{

s\_server = new SocketServer;

}

return s\_server;

}

void SocketServer::destroyInstance()

{

CC\_SAFE\_DELETE(s\_server);

}

SocketServer::SocketServer() :

\_socketServer(0),

onRecv(nullptr),

onStart(nullptr),

onNewConnection(nullptr)

{

}

SocketServer::~SocketServer()

{

this->clear();

};

void SocketServer::clear()

{

if (\_socketServer)

{

\_mutex.lock();

this->closeConnect(\_socketServer);

\_mutex.unlock();

}

for (auto msg : \_UIMessageQueue)

{

CC\_SAFE\_DELETE(msg);

}

\_UIMessageQueue.clear();

Director::getInstance()->getScheduler()->unscheduleAllForTarget(this);

}

bool SocketServer::startServer(unsigned short port)

{

if (!initServer(port))

{

return false;

}

return true;

}

bool SocketServer::initServer(unsigned short port)

{

if (\_socketServer != 0)

{

this->closeConnect(\_socketServer);

}

\_socketServer = socket(AF\_INET, SOCK\_STREAM, 0);

if (error(\_socketServer))

{

log("socket error!");

\_socketServer = 0;

return false;

}

do

{

struct sockaddr\_in sockAddr;

memset(&sockAddr, 0, sizeof(sockAddr));

\_serverPort = port; // save the port

sockAddr.sin\_family = AF\_INET;

sockAddr.sin\_port = htons(\_serverPort);

sockAddr.sin\_addr.s\_addr = htonl(INADDR\_ANY);

int ret = 0;

ret = bind(\_socketServer, (const sockaddr\*)&sockAddr, sizeof(sockAddr));

if (ret < 0)

{

log("bind error!");

break;

}

ret = listen(\_socketServer, 5);

if (ret < 0)

{

log("listen error!");

break;

}

// start

char hostName[256];

gethostname(hostName, sizeof(hostName));

struct hostent\* hostInfo = gethostbyname(hostName);

char\* ip = inet\_ntoa(\*(struct in\_addr \*)\*hostInfo->h\_addr\_list);

this->acceptClient();

if (onStart != nullptr)

{

log("start server!");

this->onStart(ip);

}

return true;

} while (false);

this->closeConnect(\_socketServer);

\_socketServer = 0;

return false;

}

void SocketServer::acceptClient()

{

std::thread th(&SocketServer::acceptFunc, this);

th.detach();

}

void SocketServer::acceptFunc()

{

int len = sizeof(sockaddr);

struct sockaddr\_in sockAddr;

while (true)

{

HSocket clientSock = accept(\_socketServer, (sockaddr\*)&sockAddr, &len);

if (error(clientSock))

{

log("accept error!");

}

else

{

this->newClientConnected(clientSock);

}

}

}

void SocketServer::newClientConnected(HSocket socket)

{

log("new connect!");

\_clientSockets.push\_back(socket);

std::thread th(&SocketServer::recvMessage, this, socket);

th.detach();

if (onNewConnection)

{

std::lock\_guard<std::mutex> lk(\_UIMessageQueueMutex);

SocketMessage \* msg = new SocketMessage(NEW\_CONNECTION, (unsigned char\*)&socket, sizeof(HSocket));

\_UIMessageQueue.push\_back(msg);

}

}

void SocketServer::recvMessage(HSocket socket)

{

char buff[1024];

int ret = 0;

while (true)

{

ret = recv(socket, buff, sizeof(buff), 0);

if (ret < 0)

{

log("recv(%d) error!", socket);

break;

}

else

{

if (ret > 0 && onRecv != nullptr)

{

std::lock\_guard<std::mutex> lk(\_UIMessageQueueMutex);

RecvData recvData;

recvData.socketClient = socket;

memcpy(recvData.data, buff, ret);

recvData.dataLen = ret;

SocketMessage \* msg = new SocketMessage(RECEIVE, (unsigned char\*)&recvData, sizeof(RecvData));

\_UIMessageQueue.push\_back(msg);

}

}

}

\_mutex.lock();

this->closeConnect(socket);

if (onDisconnect != nullptr)

{

std::lock\_guard<std::mutex> lk(\_UIMessageQueueMutex);

SocketMessage \* msg = new SocketMessage(DISCONNECT, (unsigned char\*)&socket, sizeof(HSocket));

\_UIMessageQueue.push\_back(msg);

}

\_mutex.unlock();

}

void SocketServer::sendMessage(HSocket socket, const char\* data, int count)

{

for (auto& sock : \_clientSockets)

{

if (sock == socket)

{

int ret = send(socket, data, count, 0);

if (ret < 0)

{

log("send error!");

}

break;

}

}

}

void SocketServer::sendMessage(const char\* data, int count)

{

for (auto& socket : \_clientSockets)

{

int ret = send(socket, data, count, 0);

if (ret < 0)

{

log("send error!");

}

}

}

void SocketServer::update(float dt)

{

if (\_UIMessageQueue.size() == 0)

{

return;

}

\_UIMessageQueueMutex.lock();

if (\_UIMessageQueue.size() == 0)

{

\_UIMessageQueueMutex.unlock();

return;

}

SocketMessage \*msg = \*(\_UIMessageQueue.begin());

\_UIMessageQueue.pop\_front();

switch (msg->getMsgType())

{

case NEW\_CONNECTION:

if (onNewConnection)

{

this->onNewConnection(\*(HSocket\*)msg->getMsgData()->getBytes());

}

break;

case DISCONNECT:

if (onDisconnect)

{

this->onDisconnect(\*(HSocket\*)msg->getMsgData()->getBytes());

}

break;

case RECEIVE:

if (onRecv)

{

RecvData\* recvData = (RecvData\*)msg->getMsgData()->getBytes();

this->onRecv(recvData->socketClient, (const char\*)recvData->data, recvData->dataLen);

}

break;

default:

break;

}

CC\_SAFE\_DELETE(msg);

\_UIMessageQueueMutex.unlock();

}