הקדמה

אז למה במקביל?

* עולם התכנות המקבילי התפתח בצורה מואצת בשנים האחרונות. עוד ועוד מגלים כי כאשר ממקבלים תוכניות מחשב שונות ניתן להכפיל ואף לשלש את יעילות ביצוען.
* לאחר מכן בעקבות השימוש המוגבר ברשת ומשתמשים רבים בשירותים שונים בו זמנית, התברר שהשימוש במקבילי הוא אמנם מוסיף יעילות ומגביר מהירות זמני ריצה, אך גם גורר לבעיות שהתעוררו כמו RACE-DATA ו - CONDITION-RACE שפגעו בנכונות הפעולות ויצרו פגיעה בנתונים.
* כל זאת הביא למציאת פתרונות טכנולוגים רבים למקבול תוכניות באופן שישמור על יתרונות המקבול אך גם יגן כראוי על משתנים משותפים למספר משתמשים באופן שנכונות החישוב תצא נכונה גם כאשר המשתנה מתעדכן בו זמנית .
* ניתן לראות מנגנוני סנכרון רבים במגוון שפות המאפשרים לנו שימוש בטכנולוגיה מקבילית באופן בטוח.

בואו נראה את זה בפועל...

בפרויקט זה מוצג קשר בין מצלמות להודעות – כל מצלמה מייצרת הודעה ושולחת אותה לשרת כל שלוש שניות.

בפרויקט אעסוק במערכת סימולטור להודעות מהתקן מצלמה

המערכת תכיל קלינט - סימולטור המדמה שליחת הודעות  מהתקן מצלמות  וסרבר - שיקבל את ההודעות ויציג/ישמור אותן.

קיימים מספר מצלמות וכל אחת מייצרת הודעות.

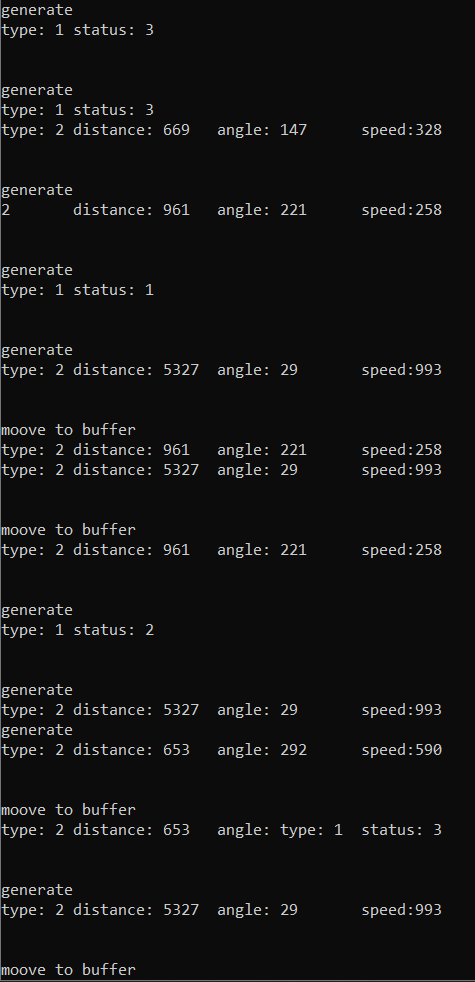
לכל מצלמה יש מערך דינאמי של הודעות שכל הזמן היא מייצרת וכל שלוש שניות ההודעה נשלחת לשרת.

כל מצלמה מייצרת הודעה ובמקבליות גם שולחת את ההודעה לשרת.

כמו כן מוצג גם סוג כל הודעה שנשלחת לשרת וכן את פרטי ההודעה.

והכל קורה אוטומטי המצלמות מייצרות הודעות ושולחות אותן למערך ההודעות שלהן ומשם תוך כדי יצירה נשלחת ההודעה המוכנה לשרת. ע"י מקביליות של מערך המצלמות הקיימות וכן של הפונקציה השולחת את ההודעות לשרת.

מדריך למשתמש



תיאור המחלקות:

Server:

* StatusMesseage
* Simulator
* Simulatir
* DiscoverMessege
* Camera
* Buffer
* baseMessage

statuseMesseage מחלקת

|  |  |
| --- | --- |
| **שדה** | **פירוט** |
| unsigned char\* messageBuffer; | תוכן ההודעה |
| int messageId; | קוד ההודעה |
| int messageType; | סוג ההודעה |
| virtual void parseMessage() = 0; |  |
| virtual void parseBack() = 0; |  |
| virtual void print()=0; | מדפיסה את ההודעה |
| unsigned char\* getMessageBuffer(); | מחזיר את תוכן ההודעה |
| baseMessage(int messageId,int messageType); | מחזירה את סוג ההודעה |

simulator מחלקת

|  |  |
| --- | --- |
| **שדה** | **פירוט** |
| Camera\*\* arrCamera; | מערך מצלמות |
| void ThreadRunCamera(); | יצירת מערך סרדים ומילוי במצלמות |
| void printValueAllCamera(); | הדפסת תוכן הודעות המצלמות |

camera מחלקת

|  |  |
| --- | --- |
| **שדה** | **פירוט** |
| char cameraId; | קוד מצלמה |
| baseMessage\*\* bufferMessage=NULL; | מערך ההודעה למצלמה |
| int indexMessage; | אינדס |
| int numOfMessege = 0; | מספר ההודעות |
| Buffer\* buffer; | ההודעה |
| bool isActive; | האם המצלמה פעילה |
| Camera(char CameraId); |  |
| void generate(); | יצירת הודעה |
| void sendToBuffer(); | שולחת את ההודעה למערך הודעות |
| void run(); | יצירת הודעה ושליחתה למערך ההודעות |
| void stop(); | כיבוי המצלמה |
| unsigned char\*\* getBufferValue(); | קבלת מערך ההודעות |
| int getOfNumOfMessege(); | קבלת מספר ההודעות |
| void sendToServer(); | שליחה לשרת כל שלוש שניות |
| void sendToServerNode(); | שליחה הודעה לשרת |

Buffer::Buffer()

{

buffer = NULL;

count = 0;

}

void Buffer::addToBuffer(unsigned char\* newMessege)

{

m.lock();

buffer = (unsigned char\*\*)realloc(buffer, sizeof(unsigned char\*) \* (++count));

buffer[count - 1] = newMessege;

m.unlock();

}

unsigned char\*\* Buffer::getBuffer()

{

return buffer;

}

void Buffer::cleanBuffer()

{

m.lock();

free(buffer);

count = 0;

m.unlock();

}

int Buffer::getCount()

{

return this->count;

}

using namespace std;

using namespace std::literals::chrono\_literals;

Camera::Camera(char cameraId)

{

this->cameraId = cameraId;

indexMessage = -1;

bufferMessage = NULL;

buffer = new Buffer();

isActive = true;

std::mutex mutexBuffer;

}

StatusMesseage\* createStatusMessage()

{

static int id = 0;

id++;

return new StatusMesseage(id, 1, rand() % 3 + 1);

}

DiscoverMessege\* createDiscoverMessage()

{

static int id = 99;

id++;

return new DiscoverMessege(id, 2, (float)(rand() % 9500 + 500), (float)(rand() % 361), (float)(rand() % 1001));

}

Camera::~Camera()

{

free(buffer);

free(bufferMessage);

}

void Camera::generate()

{

int count = rand() % 6 + 1;

baseMessage\* base;

for (int i = 0; i < count; i++)

{

std::cout << "generate " << std::endl;

bufferMessage = (baseMessage\*\*)realloc(bufferMessage, sizeof(baseMessage\*) \* (++indexMessage + 1));

(rand() % 2 + 1 == 1) ? base = createStatusMessage() : base= createDiscoverMessage();

bufferMessage[indexMessage] = base;

bufferMessage[i]->print();

std::cout << "\n" << std::endl;

}

}

void Camera::sendToBuffer()

{

std::cout << "moove to buffer" << std::endl;

for (int i = 0; i <= indexMessage; i++)

{

bufferMessage[i]->parseBack();

buffer->addToBuffer(bufferMessage[i]->getMessageBuffer());

//bufferMessage[i]->~baseMessage();

bufferMessage[i]->print();

free(bufferMessage[i]);

}

free(bufferMessage);

bufferMessage = NULL;

numOfMessege += indexMessage;

indexMessage = -1;

}

void Camera::run()

{

while (isActive)

{

generate();

sendToBuffer();

}

}

void Camera::stop()

{

isActive = false;

}

unsigned char\*\* Camera::getBufferValue()

{

return buffer->getBuffer();

}

int Camera::getOfNumOfMessege()

{

return numOfMessege;

}

void Camera::sendToServer()

{

while (isActive)

{

std::this\_thread::sleep\_for(3s);

std::cout << "send to server" << std::endl;

//buffer->cleanBuffer();

sendToServerNode();

}

}

void Camera::sendToServerNode()

{

WSAData wsaData;

WORD DllVersion = MAKEWORD(2, 1);

if (WSAStartup(DllVersion, &wsaData) != 0) {

cout << "Winsock Connection Failed!" << endl;

exit(1);

}

string getInput;

SOCKADDR\_IN addr;

int addrLen = sizeof(addr);

IN\_ADDR ipvalue;

addr.sin\_addr.s\_addr = inet\_addr("127.0.0.1");

addr.sin\_port = htons(3000);

addr.sin\_family = AF\_INET;

SOCKET connection = socket(AF\_INET, SOCK\_STREAM, NULL);

if (connect(connection, (SOCKADDR\*)&addr, addrLen) == 0) {

cout << "Connected!" << endl;

for (int i = 0; i < buffer->getCount(); i++)

{

getInput = (char\*)(buffer->getBuffer()[i]);

send(connection, getInput.c\_str(), getInput.length(), 0);

}

/\*getline(cin, getInput);\*/

}

else {

cout << "Error Connecting to Host" << endl;

exit(1);

}

return;

}

DiscoverMessege

DiscoverMessege::DiscoverMessege(int messageId, int messageType, float distance, float angle, float speed):baseMessage(messageId,messageType)

{

if (distance >= 500 && distance <= 10000)

this->distance = distance;

else

this->distance = 0;

this->angle = (angle >= 0 && angle <= 360) ? angle : 0;

this->speed = (speed >= 0 && speed <= 1000) ? speed : 0;

}

void DiscoverMessege::parseMessage()

{

if (this->messageBuffer == NULL)

return;

std::memcpy((void\*)&(this->messageType), (void\*)this->messageBuffer, 2);

std::memcpy((void\*)&(this->distance), (void\*)(this->messageBuffer+2), 4);

std::memcpy((void\*)&(this->angle), (void\*)(this->messageBuffer+6), 4);

std::memcpy((void\*)&(this->speed), (void\*)(this->messageBuffer+10), 4);

}

void DiscoverMessege::parseBack()

{

this->messageBuffer = (unsigned char\*)malloc(14);

if (this->messageBuffer == NULL)

{

std::cout << "alocation failed\n";

exit(1);

}

std::memcpy(this->messageBuffer, &(this->messageType), 2);

std::memcpy((this->messageBuffer + 2), &(this->distance), 4);

std::memcpy((this->messageBuffer + 6), &(this->angle), 4);

std::memcpy((this->messageBuffer + 10), &(this->speed), 4);

}

void DiscoverMessege::print()

{

std::cout << "type: " << this->messageType << "\tdistance: " << this->distance << "\tangle: " << this->angle << "\tspeed:" << this->speed<<"\n";

}

unsigned char\* DiscoverMessege::getMessageBuffer()

{

return this->messageBuffer;

}

מחלקת Simulatir

StatusMesseage\* Simulatir::createStatusMessage()

{

static int id = 0;

id++;

return new StatusMesseage(id,1,rand()%3+1);

}

DiscoverMessege\* Simulatir::createDiscoverMessage()

{

static int id =99;

id++;

return new DiscoverMessege(id, 2, rand() % 9500 + 500, rand() % 361 ,rand() % 1001 );

}

void Simulatir::generateAndSendMessage()

{

std::cout << "before:\n";

for (int i = 0; i < buffer1; i++) {

(rand() % 2 + 1 == 1)? buffer[i] = createStatusMessage(): buffer[i] = createDiscoverMessage();

buffer[i]->print();

}

std::cout << "after:\n";

for (int i = 0; i < buffer1; i++) {

buffer[i]->parseBack();

buffer[i]->parseMessage();

buffer[i]->print();

buffer[i]->~baseMessage();

}

}

Simulator::Simulator()

{

arrCamera = (Camera\*\*)malloc(sizeof(Camera\*) \* NUM2);

for (int i = 0; i < NUM2; i++)

{

arrCamera[i] = new Camera(char(i+'a'));

}

}

void Simulator::ThreadRunCamera()

{

char c;

std::thread threadCameraGenerate[NUM2];

std::thread threadCameraSend[NUM2];

for (int i = 0; i < NUM2; i++)

{

threadCameraGenerate[i]= std::thread(&Camera::run, \*(arrCamera[i]));

threadCameraSend[i] = std::thread(&Camera::sendToServer, \*(arrCamera[i]));

//threadCamera[i].join();

}

std::cin >> c;

for (int i = 0; i < NUM2; i++)

{

arrCamera[i]->stop();

threadCameraGenerate[i].detach();

}

}

void Simulator::printValueAllCamera()

{

for (int i = 0; i < NUM2; i++)

{

unsigned char\*\* results = arrCamera[i]->getBufferValue();

std::cout << std::endl;

std::cout << "camera number: " << i << std::endl;

for (int j = 0; j < 4; j++)

{

std::cout << j + 1<<": " << results[j] << std::endl;

}

}

}

void StatusMesseage::parseMessage()

{

if (this->messageBuffer == NULL)

return;

std::memcpy(&(this->messageType),this->messageBuffer, 2);

std::memcpy(&(this->status),(messageBuffer + 2), 1);

}

void StatusMesseage::parseBack()

{

this->messageBuffer = (unsigned char\*)malloc(3);

std::memcpy(this->messageBuffer,&(this->messageType) , 2);

std::memcpy((messageBuffer + 2),&(this->status), 1);

}

StatusMesseage::StatusMesseage(int messageId, int messageType, short status):baseMessage(messageId, messageType)

{

if (status < 1 || status>3)

this->status = 0;

else

this->status = status;

}

void StatusMesseage::print()

{

std::cout << "type: " << this->messageType << "\t" << "status: " << this->status << "\n";

}

int main()

{

Simulator\* simulator = new Simulator();

simulator->ThreadRunCamera();

simulator->printValueAllCamera();

}