Aplikacja do zarządzania zasobami sprzętowymi

Przeznaczona do zastosowania w środowisku systemu operacyjnego Microsoft Windows w architekturze client – server z użyciem modelu TCP/IP.

Jakub Skowron
Jacek Barć
Radosław Loth

Funkcje programu

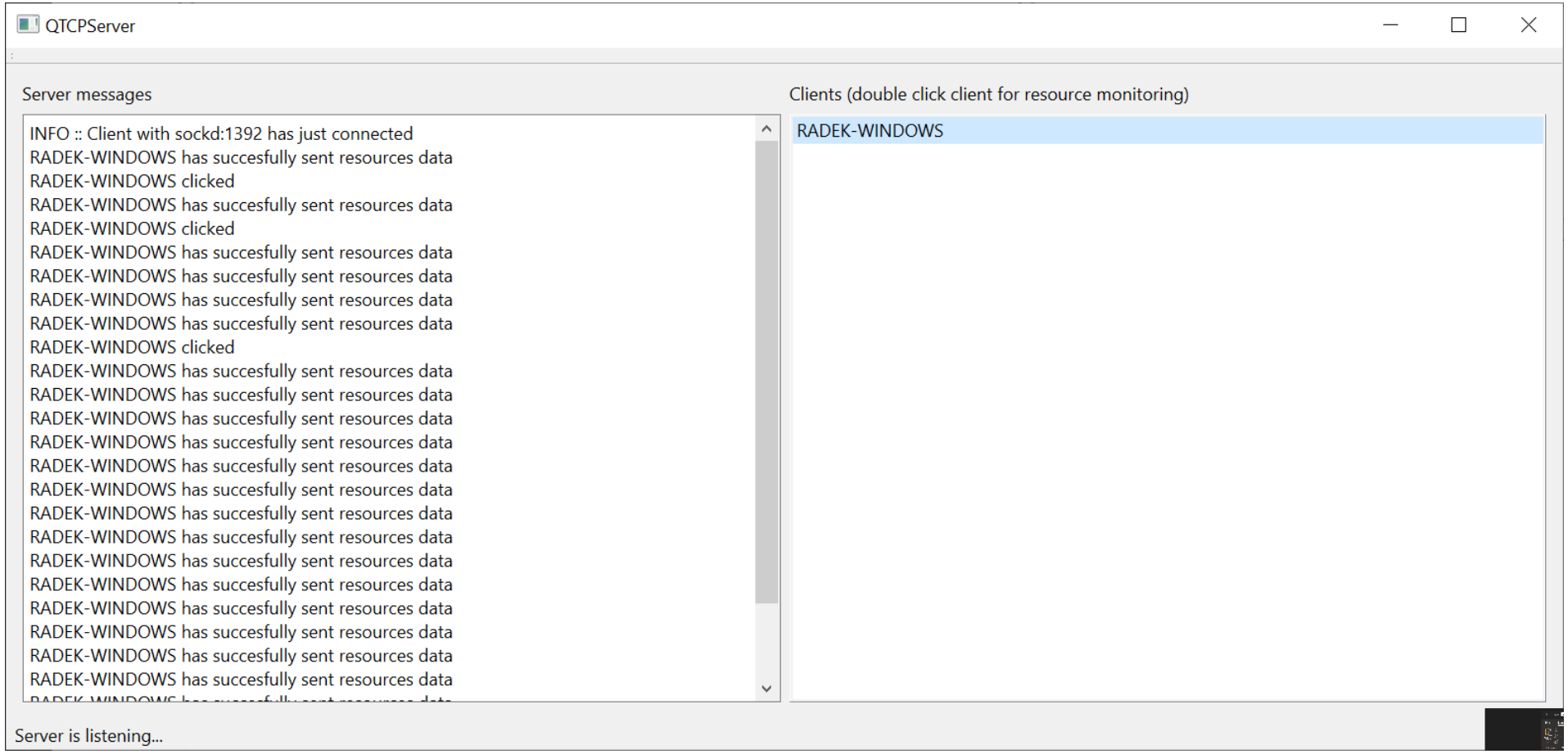
Serwer

- Nasłuchuje połączenia klienta
- Wyświetla listę klientów
- Umożliwia obserwacje zużycia zasobów sprzętowych wybranego klienta

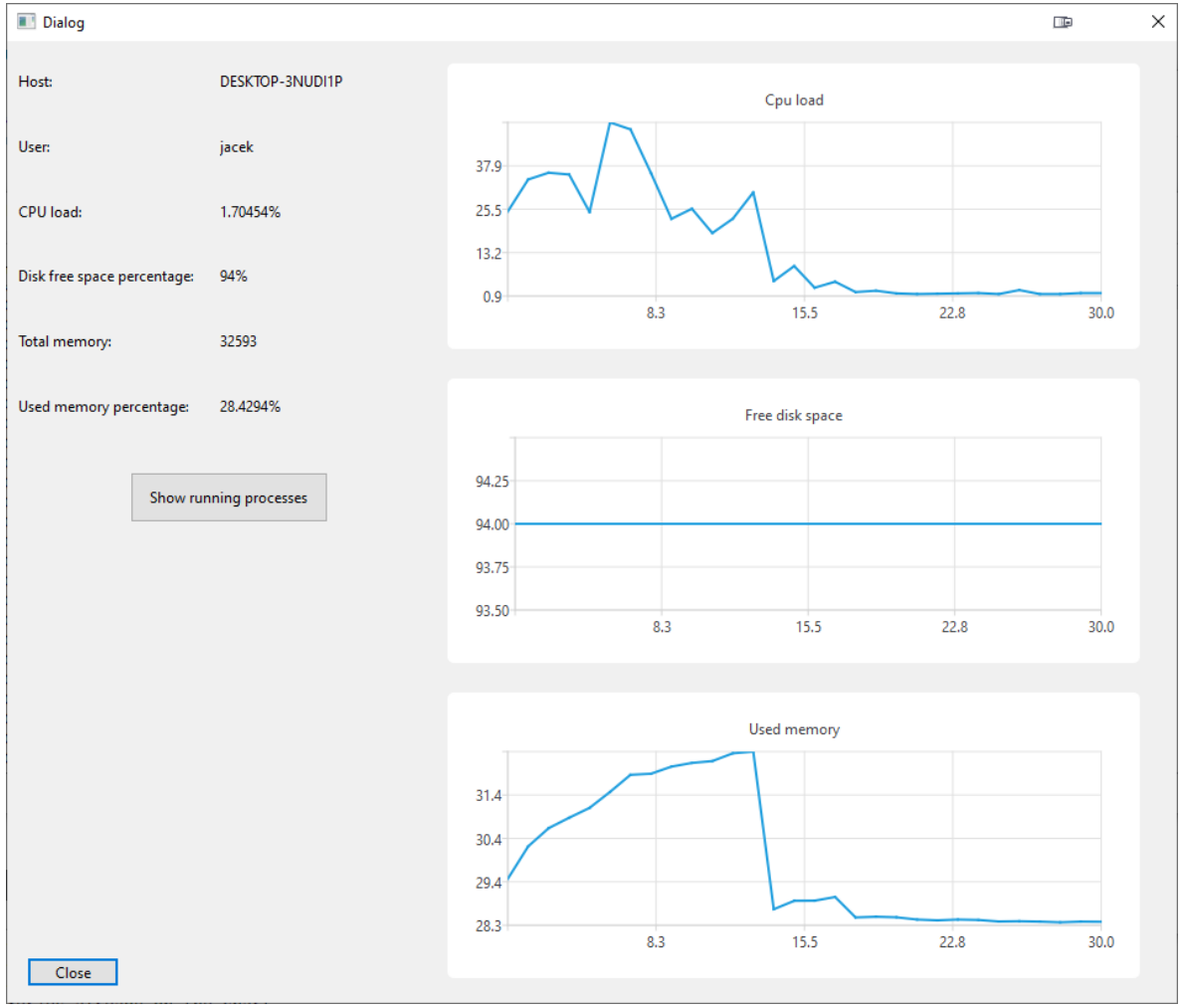
Klient

- Zbiera informacje na temat zużycia zasobów sprzętowych
- Nawiązuje połączenie z serwerem
- Wysyła informację dotyczące zużycia zasobów sprzętowych do serwera

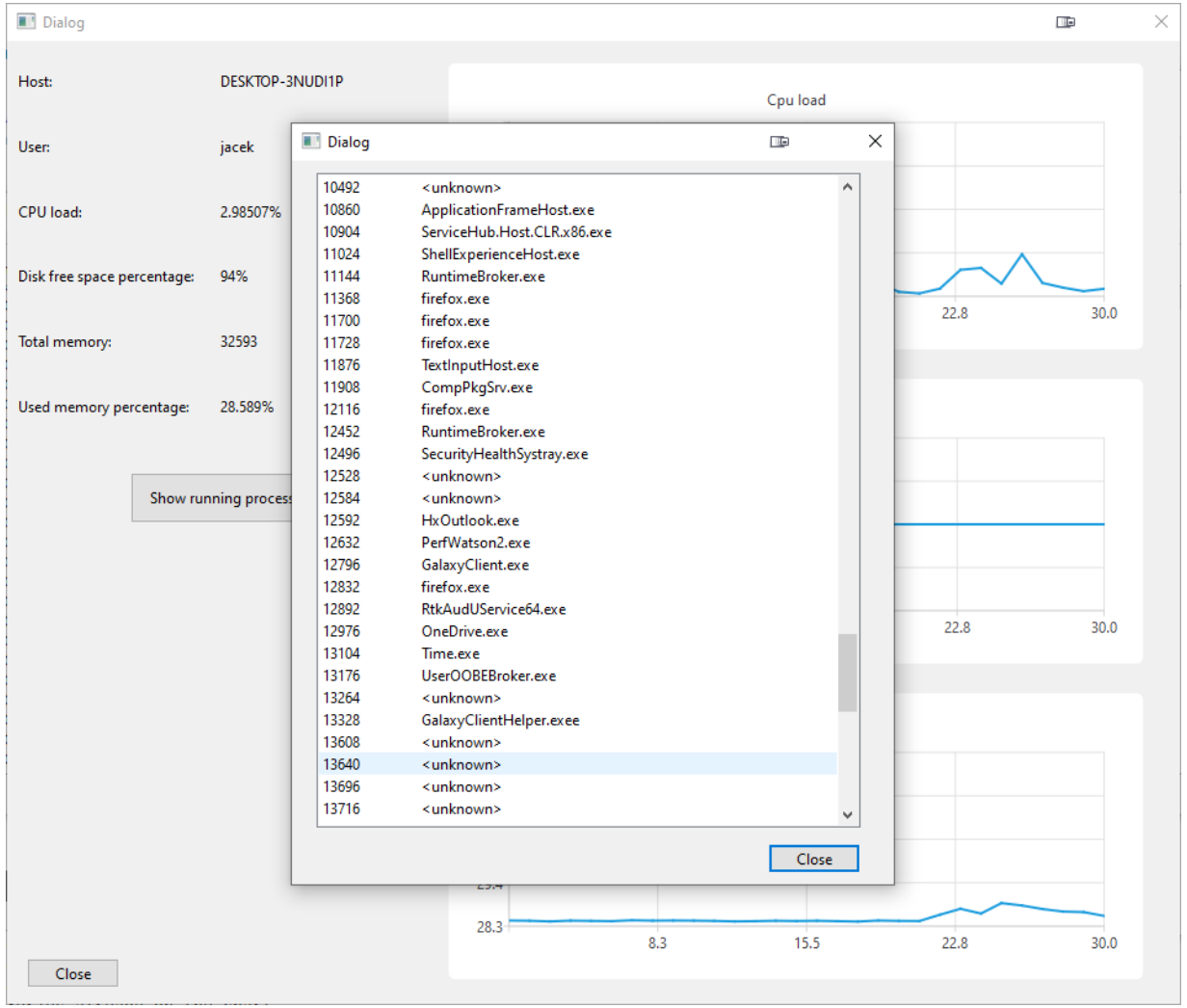
Aplikacja serwerowa



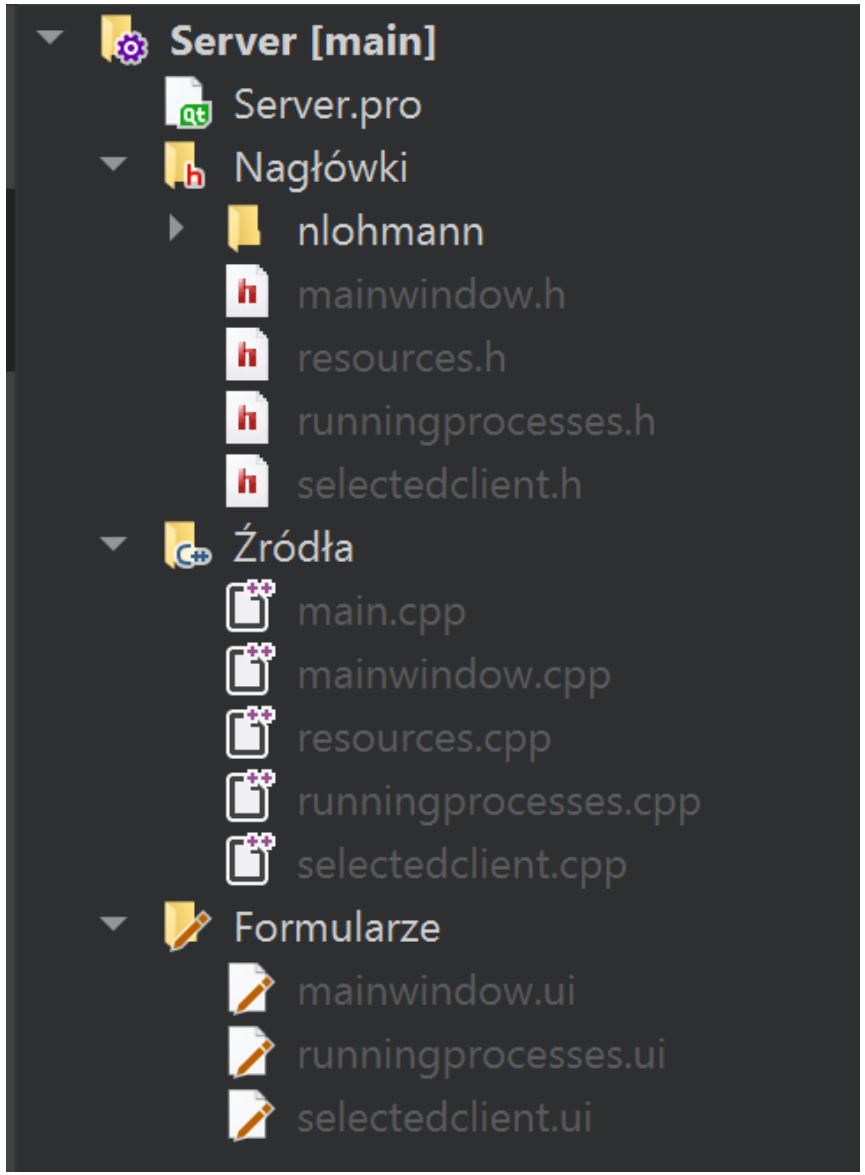
Aplikacja serwerowa



Wyświetlanie procesów klienta



Kod programu aplikacji serwerowej



Ustawienie nasłuchującego serwera

```
MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent), ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    m_server = new QTcpServer();

    if(m_server->listen(QHostAddress::Any, 6881))
    {
       connect(this, &MainWindow::newMessage, this, &MainWindow::displayMessage);
       connect(m_server, &QTcpServer::newConnection, this, &MainWindow::newConnection);
       ui->statusBar->showMessage("Server is listening...");
    }
    else
    {
        QMessageBox::critical(this,"QTCPServer",QString("Unable to start the server: %1.").arg(m_server->errorString()));
        exit(EXIT_FAILURE);
    }
}
```

Zamknięcie serwera

```
MainWindow::~MainWindow()
{
    foreach (QTcpSocket* socket, connection_set)
    {
        socket->close();
        socket->deleteLater();
    }

    m_server->close();
    m_server->deleteLater();

    delete ui;
}
```

Wyświetlanie danych klienta

```
SelectedClient::SelectedClient(MainWindow* mainWindow, Resources* r) : QDialog(nullptr),
    ui(new Ui::SelectedClient)
{
    resources = r;
    ui->setupUi(this);
    charts[0]->setTitle("Cpu load");
    charts[1]->setTitle("Free disk space");
    charts[2]->setTitle("Used memory");

    refreshView();
    QObject::connect(mainWindow, &MainWindow::newMessage, this, &SelectedClient::refreshView);
}
```

```
void SelectedClient::refreshView() {

    ui->labelHost->setText(QString::fromStdString(resources->getHostName()));
    ui->labelUser->setText(QString::fromStdString(resources->getUserName()));
    ui->labelCpuLoad->setText(QString::number(resources->getCpuLoad()) + "%");
    ui->labelDisk->setText(QString::number(resources->getDiskFreeSpacePercentage())+ "%");
    ui->labelMemory->setText(QString::number(resources->getTotalMemory()));
    ui->labelUsedMemory->setText(QString::number(100.0*resources->getMemoryLoad()/resources->getTotalMemory()) + "%");

    refreshFloatChart(ui->graphicsView_chart0, lineSeries[0], charts[0], resources->getCpuLoadList());
    refreshFloatChart(ui->graphicsView_chart1, lineSeries[1], charts[1], resources->getDiskFreeSpacePercentageList());
    refreshMemoryChart(ui->graphicsView_chart2, lineSeries[2], charts[2], resources->getMemoryLoadListReference());
}
```

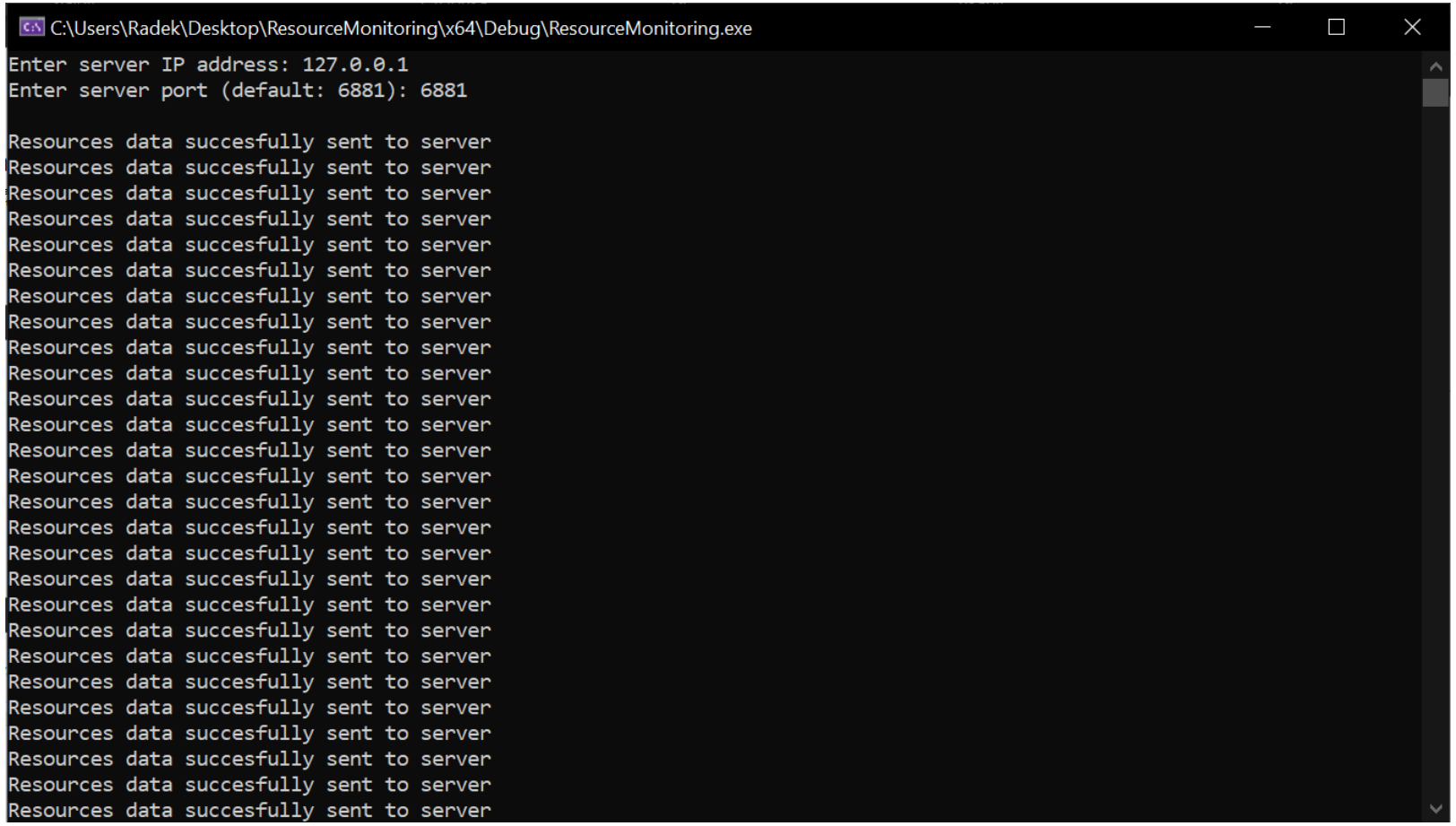
Wyświetlanie procesów

```
RunningProcesses::RunningProcesses(QWidget *parent) :
    QDialog(parent),
    ui(new Ui::RunningProcesses)
{
    ui->setupUi(this);
}

RunningProcesses::RunningProcesses(std::map<int, std::string> i_processesMap) :
    QDialog(nullptr),
    ui(new Ui::RunningProcesses)
{
    ui->setupUi(this);
    processesMap = i_processesMap;
    for (const auto& [key, value] : processesMap) {
        ui->listWidgetProcesses->addItem(QString::number(key) + " \t" + QString::fromStdString(value));
    }
}

RunningProcesses::~RunningProcesses()
{
    delete ui;
}
```

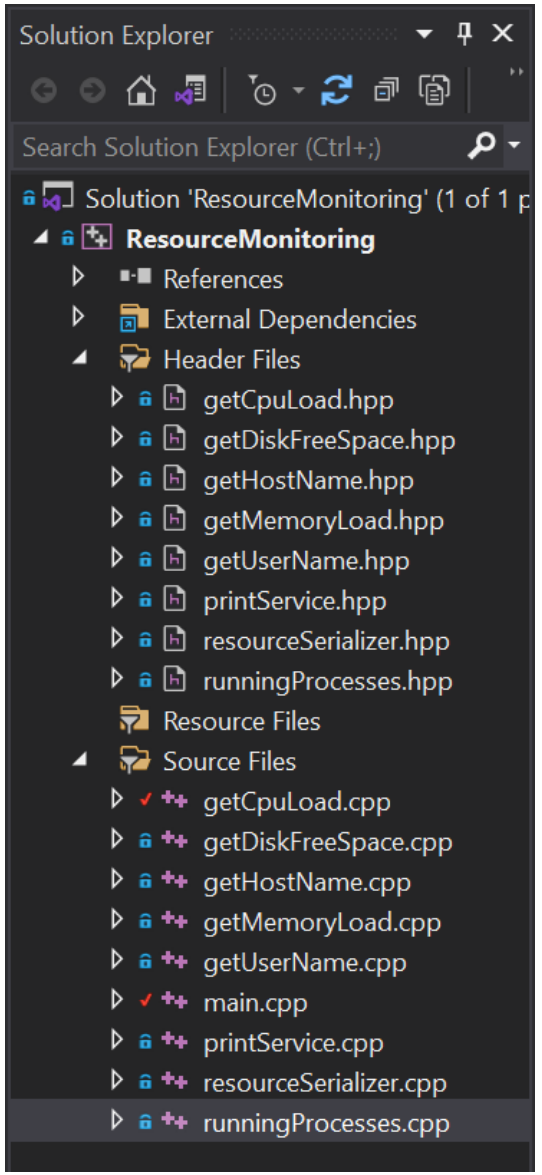
Aktualizowanie i deserializacja danych

```
void Resources::updateLists()
{
    cpuLoadList.push_front(cpuLoad);
    cpuLoadList.pop_back();
    diskFreeSpacePercentageList.push_front(diskFreeSpacePercentage);
    diskFreeSpacePercentageList.pop_back();
    memoryLoadList.push_front(memoryLoad);
    memoryLoadList.pop_back();
}

Resources::Resources()
{
    for (int i =0; i < 30; i++){
        cpuLoadList.push_front(0.0f);
        diskFreeSpacePercentageList.push_front(0.0f);
        memoryLoadList.push_front(0.0f);
    }
}

void Resources::DeserializeJson(json resourcesJson)
{
    hostName = resourcesJson["hostName"];
    userName = resourcesJson["userName"];
    totalMemory = resourcesJson["totalMemory"];
    memoryLoad = resourcesJson["memoryLoad"];
    diskFreeSpacePercentage = resourcesJson["diskFreeSpacePercentage"];
    cpuLoad = resourcesJson["cpuLoad"];
    processesMap = resourcesJson["processesMap"].get<std::map<int, std::string>>();
    updateLists();
}
```

Aplikacja kliencka



Kod programu aplikacji klienckiej



Obliczanie użycia procesora

```cpp
#include "getCpuLoad.hpp"

static float CalculateCPULoad(unsigned long long idleTicks, unsigned long long totalTicks){
    static unsigned long long _previousTotalTicks = 0;
    static unsigned long long _previousIdleTicks = 0;

    unsigned long long totalTicksSinceLastTime = totalTicks - _previousTotalTicks;
    unsigned long long idleTicksSinceLastTime = idleTicks - _previousIdleTicks;

    float ret = 1.0f - ((totalTicksSinceLastTime > 0) ? ((float)idleTicksSinceLastTime) / totalTicksSinceLastTime : 0);

    _previousTotalTicks = totalTicks;
    _previousIdleTicks = idleTicks;
    return ret;
}

static unsigned long long FileTimeToInt64(const FILETIME& ft) { return (((unsigned long long)(ft.dwHighDateTime)) << 32) | ((unsigned long long)ft.dwLowDateTime); }

float GetCPULoad(){
    FILETIME idleTime, kernelTime, userTime;
    return GetSystemTimes(&idleTime, &kernelTime, &userTime) ? CalculateCPULoad(FileTimeToInt64(idleTime), FileTimeToInt64(kernelTime) + FileTimeToInt64(userTime)) : -1.0f;
}
```

Z uwagi na to że procesor działa w sposób 0/1 z częstotliwością swojego zegara to, obliczanie użycia procesora polega na zmierzeniu czasu w którym procesor pozostaje w stanie spoczynku. Stosunek całkowitego czasu do czasu spoczynku daje wartość procentową wykorzystania procesora.

Obliczanie użycia dysku twardego

```cpp
#include "getDiskFreeSpace.hpp"

int getDiskFreeSpacePercentage()
{
    DWORD lpSectorsPerCluster,
        lpBytesPerSector,
        lpNumberOfFreeClusters,
        lpTotalNumberOfClusters;

    if (GetDiskFreeSpace(NULL,
        &lpSectorsPerCluster,
        &lpBytesPerSector,
        &lpNumberOfFreeClusters,
        &lpTotalNumberOfClusters))
    {
        return int(double(lpNumberOfFreeClusters) / double(lpTotalNumberOfClusters) * 100.0);
    }
    else
    {
        return 0;
    }
}
```

Obliczanie użycia dysku twardego polega na obliczeniu stosunku wszystkich sektorów pamięci do wolnych sektorów pamięci.

Obliczanie użycia pamięci operacyjnej

```cpp
#include "getMemoryLoad.hpp"

enum class memoryType {
    totalVirtualMem,
    totalPhysMem,
    virtualMemUsed,
    physMemUsed
};

DWORDLONG getMemoryLoad(memoryType type) {
    MEMORYSTATUSEX memInfo;
    memInfo.dwLength = sizeof(MEMORYSTATUSEX);
    GlobalMemoryStatusEx(&memInfo);

    //  Virtual memory
    DWORDLONG totalVirtualMem = memInfo.ullTotalPageFile;
    //  Physical memory
    DWORDLONG totalPhysMem = memInfo.ullTotalPhys;
    //  Virtual memory used
    DWORDLONG virtualMemUsed = memInfo.ullTotalPageFile - memInfo.ullAvailPageFile;
    //  Physical memory used
    DWORDLONG physMemUsed = memInfo.ullTotalPhys - memInfo.ullAvailPhys;

    return (type == memoryType::totalVirtualMem) ? totalVirtualMem : (type == memoryType::totalPhysMem) ? totalPhysMem :
        (type == memoryType::virtualMemUsed) ? virtualMemUsed : (type == memoryType::physMemUsed) ? physMemUsed : DWORDLONG("ERROR");
}

DWORDLONG getTotalVirtualMemory() {
    return (getMemoryLoad(memoryType::totalVirtualMem) /1024) /1024;
}
DWORDLONG getTotalPhysicalMemory() {
    return (getMemoryLoad(memoryType::totalPhysMem) /1024) /1024;
}
DWORDLONG getVirtualMemoryLoad() {
    return (getMemoryLoad(memoryType::virtualMemUsed) /1024) /1024;
}
DWORDLONG getPhysicalMemoryLoad() {
    return (getMemoryLoad(memoryType::physMemUsed) /1024) /1024;
}
```

Obliczanie użycia pamięci operacyjnej polega na pobraniu informacji dotyczącej ogółu pamięci w systemie oraz wolnej pamięci w systemie. Na podstawie tych dwóch wartości możemy się dowiedzieć ile procent pamięci jest w użyciu.

Pobieranie nazwy klienta

```cpp
#include "getHostName.hpp"

std::string getHostName()
{
    const int INFO_BUFFER_SIZE = 32767;
    TCHAR  infoBuff[INFO_BUFFER_SIZE];
    DWORD  bufCharCount = INFO_BUFFER_SIZE;
    if (!GetComputerName(infoBuff, &bufCharCount))
    {
        return "error!";
    }
    else
    {
        std::wstring wideString(&infoBuff[0]);
        std::string nString(wideString.begin(), wideString.end());
        return nString;
    }
}
```

Pobieranie nazwy użytkownika

```cpp
#include "getUserName.hpp"

std::string getUserName()
{
    const int INFO_BUFFER_SIZE = 32767;
    TCHAR  infoBuff[INFO_BUFFER_SIZE];
    DWORD  bufCharCount = INFO_BUFFER_SIZE;
    if (!GetUserName(infoBuff, &bufCharCount))
    {
        return "error!";
    }
    else
    {
        std::wstring wideString(&infoBuff[0]);
        std::string nString(wideString.begin(), wideString.end());
        return nString;
    }
}
```

Serializacja danych

```cpp
#include "resourceSerializer.hpp"

void ResourceSerializer::updateResources()
{
    resources["hostName"] = getHostName();
    resources["userName"] = getUserName();
    resources["cpuLoad"] = GetCPULoad();
    resources["totalMemory"] = getTotalPhysicalMemory();
    resources["memoryLoad"] = getPhysicalMemoryLoad();
    resources["diskFreeSpacePercentage"] = getDiskFreeSpacePercentage();
    RunningProcesses processes;
    std::string** processesArr = processes.getRunningProcessesArray();
    std::map<int, std::string> processesMap;
    int n = processes.getRuninigProcessesNumber();
    for (int i = 0; i < n; i++)
    {
        remove(processesArr[i][1].begin(), processesArr[i][1].end(), ' ');
        processesMap.insert({ atoi(processesArr[i][0].c_str()), processesArr[i][1] });
    }
    for (int i = 0; i < n; i++)
    {
        processesArr[i] = nullptr;
        delete[] processesArr[i];
    }
    resources["processesMap"] = processesMap;
}

ResourceSerializer::ResourceSerializer()
{
    updateResources();
}

json ResourceSerializer::getResourcesJson()
{
    updateResources();
    return resources;
}
```

Pobieranie listy uruchomionych procesów.

```cpp
#include "runningProcesses.hpp"

RunningProcesses::RunningProcesses() { findRunningProcesses(); }

RunningProcesses::~RunningProcesses() {
    for (int i = 0; i < runinigProcessesNumber; i++)
    {
        runningProcessesArray[i] = nullptr;
        delete[] runningProcessesArray[i];
    }
    delete[] runningProcessesArray;
}

int RunningProcesses::getRuninigProcessesNumber() { return runinigProcessesNumber; }

std::string** RunningProcesses::getRunningProcessesArray() { return runningProcessesArray; }

void RunningProcesses::findRunningProcesses()
{
    DWORD aProcesses[1024], cbNeeded, cProcesses;
    if (!EnumProcesses(aProcesses, sizeof(aProcesses), &cbNeeded))
    {
        runinigProcessesNumber = 0;
        return;
    }

    cProcesses = cbNeeded / sizeof(DWORD);
    runinigProcessesNumber = cProcesses;
    runningProcessesArray = new std::string * [runinigProcessesNumber];
    for (int i = 0; i < cProcesses; i++)
    {
        runningProcessesArray[i] = new std::string[2];
        std::ostringstream stream;
        stream << aProcesses[i];
        runningProcessesArray[i][0] = stream.str();
        if (aProcesses[i] != 0)
        {
            runningProcessesArray[i][1] = findProcessName(aProcesses[i]);
        }
        else
        {
            runningProcessesArray[i][1] = "system";
        }
    }
}
```

```cpp
std::string RunningProcesses::findProcessName(DWORD processID)
{
    TCHAR szProcessName[MAX_PATH] = TEXT("<unknown>");
    HANDLE hProcess = OpenProcess(PROCESS_QUERY_LIMITED_INFORMATION | PROCESS_VM_READ, FALSE, processID);
    if (NULL != hProcess)
    {
        HMODULE hMod;
        DWORD cbNeeded;
        if (EnumProcessModulesEx(hProcess, &hMod, sizeof(hMod), &cbNeeded, LIST_MODULES_ALL))
        {
            GetModuleBaseName(hProcess, hMod, szProcessName, sizeof(szProcessName) / sizeof(TCHAR));
        }
    }
    CloseHandle(hProcess);
    using convert_type = std::codecvt_utf8<wchar_t>;
    std::wstring_convert<convert_type, wchar_t> converter;
    std::string returnString = converter.to_bytes((std::wstring)szProcessName);
    return returnString;
}
```

Połączenie z serwerem

```
// Initialize Winsock
iResult = WSAStartup(MAKEWORD(2, 2), &wsaData);
if (iResult != 0) {
    printf("WSAStartup failed with error: %d\n", iResult);
    return 1;
}

ZeroMemory(&hints, sizeof(hints));
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
hints.ai_protocol = IPPROTO_TCP;

std::cout << "Enter server IP address: ";
std::getline(std::cin, ipAddress);
std::cout << "Enter server port (default: 6881): ";
std::getline(std::cin, port);
std::cout << std::endl;

// Resolve the server address and port
iResult = getaddrinfo(ipAddress.c_str(), port.c_str(), &hints, &result);
if (iResult != 0) {
    printf("getaddrinfo failed with error: %d\n", iResult);
    WSACleanup();
    return 1;
}

// Attempt to connect to an address until one succeeds
for (ptr = result; ptr != NULL; ptr = ptr->ai_next) {

    // Create a SOCKET for connecting to server
    ConnectSocket = socket(ptr->ai_family, ptr->ai_socktype,
        ptr->ai_protocol);
    if (ConnectSocket == INVALID_SOCKET) {
        printf("socket failed with error: %ld\n", WSAGetLastError());
        WSACleanup();
        return 1;
    }

    // Connect to server.
    iResult = connect(ConnectSocket, ptr->ai_addr, (int)ptr->ai_addrlen);
    if (iResult == SOCKET_ERROR) {
        closesocket(ConnectSocket);
        ConnectSocket = INVALID_SOCKET;
        continue;
    }
    break;
}
```

```
freeaddrinfo(result);

if (ConnectSocket == INVALID_SOCKET) {
    printf("Unable to connect to server!\n");
    WSACleanup();
    return 1;
}

// Send until the peer closes the connection
do {
    std::string serializedString = serializer.getResourcesJson().dump();
    sendbuf = serializedString.c_str();
iResult = send(ConnectSocket, serializedString.c_str(), serializedString.length(), 0);
    if (iResult == SOCKET_ERROR) {
        printf("send failed with error: %d\n", WSAGetLastError());
        closesocket(ConnectSocket);
        WSACleanup();
        return 1;
    }
    if (iResult > 0)
        printf("Resources data succesfully sent to server\n", iResult);
    else if (iResult == 0)
        printf("Connection closed\n");
    else
        printf("send failed with error: %d\n", WSAGetLastError());
    Sleep(FREQUENCY);
} while (iResult > 0);


// cleanup
closesocket(ConnectSocket);
WSACleanup();

return 0;
```