**SORUCE CODE LISTING (Without UI initial version)**

1. Server Side Code

ServerCommon.h

#pragma once

// action constants

#define ACTION\_INSTALL 1

#define ACTION\_UNINSTALL 2

#define ACTION\_COPY 3

#define ACTION\_GETINFO 4

#define ACTION\_EXIT 9

#define ACTION\_NONE 8

#define ACTION\_INVALID -9

#define ACTION\_STATUS\_COMPLETE 10

#define ACTION\_STATUS\_INPROGRESS 11

#define ACTION\_STATUS\_NEW 13

#define RSMC\_ERROR 1

#define RSMC\_SUCCESS 0

#define RSMC\_ERROR\_MEM 2

#define MAX\_APP\_NAME 50

#define MAX\_VERSION\_STRING\_SIZE 30

#define MAX\_ACTION\_STRING 20

#define MAX\_RESPONSE\_STRING 100

#define CLIENT\_REGISTERED 1

#define CLIENT\_DEREGISTERED 2

#define CLIENT\_ERROR 3

#define JOB\_STATUS\_NEW 15

#define JOB\_STATUS\_COMPLETE 16

#define JOB\_STATUS\_NONE 17

#define JOB\_STATUS\_INPROGRESS 18

#define MAX\_CLIENTS 100

struct \_JobInfo

{

int jobid;

int action;

char app[MAX\_APP\_NAME + 1];

char version[MAX\_VERSION\_STRING\_SIZE + 1];

char subaction[MAX\_ACTION\_STRING + 1];

char subaction\_arg[MAX\_ACTION\_STRING + 1];

char \*GetInfo\_Output;

long GetInfo\_Output\_size;

char action\_status\_string[MAX\_RESPONSE\_STRING + 1];

int action\_status; // RSMC\_SUCCESS or RSMC\_ERROR

int action\_retval;

char response\_stringReceiveed[MAX\_RESPONSE\_STRING + 1];

int final\_job\_status;

};

typedef struct \_JobInfo CMDINFO, JOB, \*PJOB;

struct \_ClientInfo

{

char ClientIP[16];

char HostName[20];

char OSName[50];

char OSVersion[20];

int CurrentJobId;

int RegistrationStatus;

int JobStatus;

JOB Job;

};

typedef \_ClientInfo CLIENTINFO, \*PCLIENTINFO;

struct \_ServerInfo

{

char ServerIP[16];

u\_short CommPort;

u\_short StatusPort;

};

typedef \_ServerInfo SERVERINFO, \*PSERVERINFO;

int GetIPAddress(SOCKET s, char \*IP);

int GetHostName(SOCKET s, char \*IP, char \*HostName);

int RegisterClient(char \*IP, char \*OsName, char \*OsVersion);

DWORD WINAPI ServiceWorkerThread\_ManageJostStatus(LPVOID lpParam);

DWORD WINAPI ServiceWorkerThread\_ManageCommunicationAndJob(LPVOID lpParam);

DWORD WINAPI ServiceWorkerThread\_UI(LPVOID lpParam);

void print(\_TCHAR \*msg);

int ProcessUIAction\_Install(char \*request);

int ProcessUIAction\_Uninstall(char \*request);

int ProcessUIAction\_Copy(char \*request);

int ProcessUIAction\_GetInfo(char \*request);

PCLIENTINFO GetClientRecord(const char \*IP);

ServerCommon.cpp

#define \_WINSOCK\_DEPRECATED\_NO\_WARNINGS

#define \_CRT\_SECURE\_NO\_WARNNINGS

#include <Windows.h>

#include <tchar.h>

#include <stdio.h>

#include <iostream>

#include <string>

#include <ctime>

#include <sstream>

#include <vector>

#include <tchar.h>

#include "ServerCommon.h"

//#include "winsock.h"

using namespace std;

#ifdef SERVICE\_RSMC

extern HANDLE g\_ServiceStopEvent;

#endif

void print(\_TCHAR \*msg)

{

#ifdef CSERVICE\_RSMC

OutputDebugString(msg);

#else

\_tprintf(msg);

#endif

}

SERVERINFO ServerInfo = {

"127.0.0.1",

55555,

55556

};

// Number of clients that we can handle

CLIENTINFO Clients[100];

int TotalClientCount = 0;

char \*getinfo\_output\_ptr = NULL;

// splits a given string based on delimter provided

// e.g.: "1:appname:version" split into

// 1

// appname

// version

//

vector<string> split(const string &s, char delim) {

vector<string> result;

stringstream ss(s);

string item;

while (getline(ss, item, delim)) {

result.push\_back(item);

}

return result;

}

int GetIPAddress(SOCKET s, char \*IP)

{

sockaddr\_in SockAddr;

int addrlen = sizeof(SockAddr);

if (getpeername(s, (LPSOCKADDR)&SockAddr, &addrlen) == SOCKET\_ERROR)

{

return WSAGetLastError();

}

strcpy(IP, inet\_ntoa(SockAddr.sin\_addr));

return RSMC\_SUCCESS;

}

int GetHostName(SOCKET s, char \*IP, char \*HostName)

{

strcpy(HostName, "default host");

return RSMC\_SUCCESS;

}

char \* GetJobStatusString(int job\_status)

{

switch (job\_status)

{

case JOB\_STATUS\_NONE:

return "None";

case JOB\_STATUS\_COMPLETE:

return "Complete";

case JOB\_STATUS\_NEW:

return "Waiting";

case JOB\_STATUS\_INPROGRESS:

return "In Progress";

default:

return "UNKNOWN";

}

}

int RegisterClient(const char \*IP, const char \*Hostname, const char \*OsName, const char \*OsVersion)

{

PCLIENTINFO cp;

cp = GetClientRecord(IP);

if (cp == NULL)

{

printf("\nRegistering new client");

if (TotalClientCount < MAX\_CLIENTS)

{

printf("\nIP = [%s]", IP);

strcpy(Clients[TotalClientCount].ClientIP, IP);

strcpy(Clients[TotalClientCount].HostName, Hostname);

Clients[TotalClientCount].RegistrationStatus = CLIENT\_REGISTERED;

strcpy(Clients[TotalClientCount].OSName, OsName);

strcpy(Clients[TotalClientCount].OSVersion, OsVersion);

Clients[TotalClientCount].Job.action = ACTION\_NONE;

Clients[TotalClientCount].Job.action\_status = JOB\_STATUS\_NONE;

strcpy(Clients[TotalClientCount].Job.action\_status\_string, "None");

Clients[TotalClientCount].Job.jobid = 1;

TotalClientCount++;

return RSMC\_SUCCESS;

}

}

else

{

printf("\nUpdating the existing client");

if (TotalClientCount < MAX\_CLIENTS)

{

printf("\nIP = [%s]", IP);

strcpy(cp->ClientIP, IP);

strcpy(cp->HostName, Hostname);

cp->RegistrationStatus = CLIENT\_REGISTERED;

strcpy(cp->OSName, OsName);

strcpy(cp->OSVersion, OsVersion);

cp->Job.action = ACTION\_NONE;

cp->Job.action\_status = JOB\_STATUS\_NONE;

strcpy(cp->Job.action\_status\_string, "None");

cp->Job.jobid = 1;

return RSMC\_SUCCESS;

}

}

print(\_T("Cannot register more clients, Max Limit reached"));

return RSMC\_ERROR;

}

int FindNewJob(char \*IP, char \*job\_string)

{

PCLIENTINFO pClientInfo = GetClientRecord(IP);

if (pClientInfo == NULL)

{

sprintf(job\_string, "ClientNotFound");

return RSMC\_ERROR;

}

if (pClientInfo->JobStatus == JOB\_STATUS\_NEW &&

pClientInfo->Job.action\_status == ACTION\_STATUS\_NEW)

{

printf("\nJob Action: %d", pClientInfo->Job.action);

switch (pClientInfo->Job.action)

{

case ACTION\_INSTALL:

//jobid:action:appname:version

sprintf(job\_string, "%d:%d:%s:%s", pClientInfo->Job.jobid, pClientInfo->Job.action, pClientInfo->Job.app, pClientInfo->Job.version);

break;

case ACTION\_UNINSTALL:

//jobid:action:appname:version

sprintf(job\_string, "%d:%d:%s:%s", pClientInfo->Job.jobid, pClientInfo->Job.action, pClientInfo->Job.app, pClientInfo->Job.version);

break;

case ACTION\_COPY:

//jobid:action:filename

sprintf(job\_string, "%d:%d:%s", pClientInfo->Job.jobid, pClientInfo->Job.action, pClientInfo->Job.app);

break;

case ACTION\_GETINFO:

//jobid:action:subaction

sprintf(job\_string, "%d:%d:%s", pClientInfo->Job.jobid, pClientInfo->Job.action, pClientInfo->Job.subaction);

break;

}

pClientInfo->Job.action\_status = ACTION\_STATUS\_INPROGRESS;

pClientInfo->JobStatus = JOB\_STATUS\_INPROGRESS;

}

else

{

sprintf(job\_string, "NoNewJob");

}

return RSMC\_SUCCESS;

}

int ProcessClientRequest(SOCKET s, char \*request\_string, char \*response\_string)

{

if (strstr(request\_string, "Register?") != NULL)

{

vector<string> list = split(request\_string, ':');

char IP[16];

int err = 0;

char Hostname[20];

GetIPAddress(s, IP);

if (RegisterClient(IP, list.at(1).c\_str(), list.at(2).c\_str(), list.at(3).c\_str()) == RSMC\_SUCCESS)

{

strcpy(response\_string, "Registered");

}

else

strcpy(response\_string, "Register\_failed");

}

else if (strstr(request\_string, "NewJob?") != NULL)

{

char IP[16];

GetIPAddress(s, IP);

FindNewJob(IP, response\_string); // no need to check the returnvalue, it would set the reponse string accordingly

}

else

{

strcpy(response\_string, "UnkownRequest");

return RSMC\_ERROR;

}

return RSMC\_SUCCESS;

}

DWORD WINAPI ServiceWorkerThread\_ManageCommunicationAndJob(LPVOID lpParam)

{

print(\_T("\n In ServiceWorkerThread\_ManageCommunicationAndJob()"));

#ifdef SERVICE\_RSMC

// Periodically check if the service has been requested to stop

while (WaitForSingleObject(g\_ServiceStopEvent, 0) != WAIT\_OBJECT\_0)

#else

while (1)

#endif

{

SOCKET sockSocket, acceptSocket;

struct sockaddr\_in service;

// create socket

print(\_T("\n Create socket ..."));

sockSocket = socket(AF\_INET, SOCK\_STREAM, IPPROTO\_TCP); // create a socket

if (sockSocket == INVALID\_SOCKET) // check for errors

{

OutputDebugString(\_T("Error connecting to socket!"));

//return 0;

}

// bind to socket

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

service.sin\_family = AF\_INET;

service.sin\_port = htons(ServerInfo.CommPort);

print(\_T("\n bind to socket ..."));

if (bind(sockSocket, (SOCKADDR\*)&service, sizeof(service)) == SOCKET\_ERROR) // cheking for errors

{

print(\_T("Error binding to socket!"));

closesocket(sockSocket);

//return 0;

}

do

{

if (listen(sockSocket, 10) == SOCKET\_ERROR) // check for errors

{

print(\_T("Error at listen(): "));

closesocket(sockSocket);

break;

}

//accept connection

int servlen = sizeof(service);

print(\_T("\nMCJ Waiting for client to connect..."));

acceptSocket = accept(sockSocket, (SOCKADDR\*)&service, &servlen);

if (acceptSocket == INVALID\_SOCKET)

{

print(\_T("Error at accept(): "));

closesocket(sockSocket);

break;

}

char Buffer[512];

char RBuffer[512];

int res = recv(acceptSocket, Buffer, 512, 0);

Buffer[res] = '\0';

printf("\nrecevied data: %s", Buffer);

int retval = ProcessClientRequest(acceptSocket, Buffer, RBuffer);

printf("\nsending data: %s", RBuffer);

if (send(acceptSocket, RBuffer, strlen(RBuffer), 0) == SOCKET\_ERROR)

{

print(\_T("send error: "));

break;

}

#ifdef SERVICE\_RSMC

} while (WaitForSingleObject(g\_ServiceStopEvent, 0) != WAIT\_OBJECT\_0);

#else

} while (true);

#endif

closesocket(sockSocket);

}

print(\_T("\nOut of ServiceWorkerThread\_ManageCommunicationAndJob()"));

return ERROR\_SUCCESS;

}

PCLIENTINFO GetClientRecord(const char \*IP)

{

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

{

if ((strcmp(Clients[i].ClientIP, IP) == 0))

{

return &Clients[i];

}

}

return NULL;

}

int ProcessClientStatusUpdateRequest(SOCKET s, char \*status\_string, char \*response\_string)

{

char \*getinfo\_output = NULL;

long output\_size;

vector<string> list = split(status\_string, ':');

// find the ip addrerss of the client first

char IP[16];

GetIPAddress(s, IP);

// check if action is GETINFO and It was successful, then receive the GETINFO OUTPUT

if (atoi(list.at(1).c\_str()) == ACTION\_GETINFO)

{

if (atoi(list.at(2).c\_str()) == RSMC\_SUCCESS)

{

char buffer[MAX\_RESPONSE\_STRING + 1];

int rec = recv(s, buffer, MAX\_RESPONSE\_STRING, MSG\_PEEK);

if (WSAGetLastError() == WSAEMSGSIZE)

{

}

if (rec > 0)

{

getinfo\_output = (char\*)malloc(rec);

output\_size = rec;

if (getinfo\_output == NULL)

{

return RSMC\_ERROR\_MEM;

}

int rec2 = recv(s, getinfo\_output, rec, 0);

if (!(rec2 > 0))

{

strcpy(response\_string, "GetInfoDataMissing");

return RSMC\_ERROR;

}

char filename[50];

sprintf(filename, "c:\\repos\\getinfo\_%s\_.out", IP );

FILE \*fp = fopen(filename, "w");

if (fp != NULL)

{

int size = rec;

char \* s = getinfo\_output;

while (size > 0)

{

fputc(\*s,fp);

s++;

size--;

}

}

fclose(fp);

}

}

}

PCLIENTINFO pClient = GetClientRecord(IP);

if ( pClient == NULL)

{

strcpy(response\_string, "ClientNotFound");

return RSMC\_ERROR;

}

int tjobid = atoi(list.at(0).c\_str());

if (tjobid != pClient->Job.jobid)

{

strcpy(response\_string, "JobNotFound");

return RSMC\_ERROR;

}

pClient->JobStatus = JOB\_STATUS\_COMPLETE;

pClient->Job.action\_status = ACTION\_STATUS\_COMPLETE;

pClient->Job.action\_retval = atoi(list.at(2).c\_str());

strcpy(pClient->Job.action\_status\_string, list.at(3).c\_str());

if (pClient->Job.action == ACTION\_GETINFO)

{

pClient->Job.GetInfo\_Output = getinfo\_output;

pClient->Job.GetInfo\_Output\_size = output\_size;

}

strcpy(response\_string, "ThankYou");

return RSMC\_SUCCESS;

}

int SendToClient(SOCKET s, char \*response)

{

std::string message = response;

print(\_T("\nSending response ..."));

if (send(s, message.c\_str(), message.length(), 0) == SOCKET\_ERROR)

{

//std::cout << "Error at send(): " << WSAGetLastError() << std::endl;

print(\_T("Errot sending message."));

return RSMC\_ERROR;

}

print(\_T("Done.\n"));

return RSMC\_SUCCESS;

}

DWORD WINAPI ServiceWorkerThread\_ManageJostStatus(LPVOID lpParam)

{

SOCKET sockSocket, acceptSocket;

struct sockaddr\_in service;

print(\_T("\nIn ServiceWorkerThread\_ManageJostStatus()"));

#ifdef SERVICE\_RSMC

// Periodically check if the service has been requested to stop

while (WaitForSingleObject(g\_ServiceStopEvent, 0) != WAIT\_OBJECT\_0)

#else

while (true)

#endif

{

// create socket

print(\_T("\nCreate socket ..."));

sockSocket = socket(AF\_INET, SOCK\_STREAM, IPPROTO\_TCP); // create a socket

if (sockSocket == INVALID\_SOCKET) // check for errors

{

print(\_T("Error connecting to socket!"));

//return 0;

}

// bind to socket

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

service.sin\_family = AF\_INET;

service.sin\_port = htons(55556);

print(\_T("\nbind to socket ..."));

if (bind(sockSocket, (SOCKADDR\*)&service, sizeof(service)) == SOCKET\_ERROR) // cheking for errors

{

print(\_T("Error binding to socket!"));

closesocket(sockSocket);

return 0;

}

do

{

// listen

print(\_T("\nlisten to socket ..."));

if (listen(sockSocket, 10) == SOCKET\_ERROR) // check for errors

{

print(\_T("Error at listen(): "));

closesocket(sockSocket);

return 0;

}

//accept connection

int servlen = sizeof(service);

print(\_T("\nMJS-Waiting for client to connect..."));

acceptSocket = accept(sockSocket, (SOCKADDR\*)&service, &servlen);

if (acceptSocket == INVALID\_SOCKET)

{

print(\_T("Error at accept(): "));

closesocket(sockSocket);

return 0;

}

char StatusBuffer[MAX\_RESPONSE\_STRING+1];

char ResponseString[MAX\_RESPONSE\_STRING + 1];

do

{

print(\_T("\nClient connected ...,receiving data..."));

int res = recv(acceptSocket, StatusBuffer, MAX\_RESPONSE\_STRING, 0);

if (res > 0) {

print(\_T("recevied job status!"));

break;

}

} while (true);

ProcessClientStatusUpdateRequest(acceptSocket, StatusBuffer, ResponseString);

SendToClient(acceptSocket, ResponseString);

#ifdef SERVICE\_RSMC

} while (WaitForSingleObject(g\_ServiceStopEvent, 0) != WAIT\_OBJECT\_0);

#else

} while (true);

#endif

closesocket(sockSocket);

}

return RSMC\_SUCCESS;

}

#define PIPE\_NAME "\\\\.\\pipe\\rsmcuipipe"

#define PIPE\_BUFFSIZE 512

int CreateUIPIPE(HANDLE \*hPipe)

{

\*hPipe = CreateNamedPipe(

PIPE\_NAME, // pipe name

PIPE\_ACCESS\_DUPLEX, // read/write access

PIPE\_TYPE\_BYTE | PIPE\_READMODE\_BYTE | PIPE\_WAIT, // blocking mode // message-read mode // message type pipe

PIPE\_UNLIMITED\_INSTANCES, // max. instances

PIPE\_BUFFSIZE, // output buffer size

PIPE\_BUFFSIZE, // input buffer size

0, // client time-out

NULL); // default security attribute

if (hPipe == INVALID\_HANDLE\_VALUE)

{

\_tprintf(TEXT("\nCreateNamedPipe failed, GLE=%d.\n"), GetLastError());

return RSMC\_ERROR;

}

return RSMC\_SUCCESS;

}

char\* GetActionString(int action)

{

switch (action)

{

case ACTION\_INSTALL:

return "INSTALL";

case ACTION\_UNINSTALL:

return "UNINSTALL";

case ACTION\_COPY:

return "COPY";

case ACTION\_GETINFO:

return "GETINFO";

case ACTION\_NONE:

return "NONE";

default:

return "UNKNOWN";

}

}

int GetAllClientsString(char \*ReplyBuffer)

{

int TotalClients = TotalClientCount;

int i = 0;

if (TotalClients > 0)

{

do

{

char Buffer[PIPE\_BUFFSIZE + 1];

if (i != 0)

{

strcat(ReplyBuffer, "|");

}

sprintf(Buffer, "%s:%s:%s:%s:N/A:N/A:%s:%s:%s", Clients[i].ClientIP, Clients[i].HostName, Clients[i].OSName,

Clients[i].OSVersion, GetActionString(Clients[i].Job.action),

GetJobStatusString(Clients[i].JobStatus), Clients[i].Job.action\_status\_string);

strcat(ReplyBuffer, Buffer);

i++;

} while (i < TotalClients);

return i;

}

strcpy(ReplyBuffer, "None");

return i;

}

DWORD WINAPI ServiceWorkerThread\_UI(LPVOID lpParam)

{

#ifdef SERVICE\_RSMC

// Periodically check if the service has been requested to stop

while (WaitForSingleObject(g\_ServiceStopEvent, 0) != WAIT\_OBJECT\_0)

#else

while (1)

#endif

{

bool fSuccess;

bool fConnected;

DWORD dataread;

DWORD datasize;

DWORD datawritten;

char RequestBuffer[PIPE\_BUFFSIZE + 1];

HANDLE hPipe;

printf("\nCreating UI Communication PIPE..");

CreateUIPIPE(&hPipe);

//wait for client to connect to pipe

printf("\nConnect to Named pipe");

fConnected = ConnectNamedPipe(hPipe, NULL);

if (fConnected == 0)

{

printf("\nConnectNamePipe() failed, Error: %d", GetLastError());

}

else

{

printf("\nUI Client conected to pipe");

memset(RequestBuffer, '\0', PIPE\_BUFFSIZE);

fSuccess = ReadFile(hPipe, RequestBuffer, PIPE\_BUFFSIZE, &dataread, NULL);

printf("\nNoted Data on pipe from UI");

if (!fSuccess || dataread == 0)

{

if (GetLastError() == ERROR\_BROKEN\_PIPE)

{

printf("\nUI Client disconnected.");

}

else

{

printf("\nPipe read failed. %d", GetLastError());

}

}

else

{

char \* ReplyBufferFull = (char\*)malloc(1024\*4);

memset(ReplyBufferFull, '\0', 1024 \* 4);

if (ReplyBufferFull == NULL)

{

printf("\nRun out of memory!");

exit(1);

}

printf("\nRead from Pipe: %s", RequestBuffer);

if (strstr(RequestBuffer, "refresh") )

{

// UI asking for refresh

printf("\nREFRESH request from UI");

printf("\nBefore: [%s]", ReplyBufferFull);

GetAllClientsString(ReplyBufferFull);

printf("\nData to UI:[%s]", ReplyBufferFull);

datasize = strlen(ReplyBufferFull);

//Write to fiile

fSuccess = WriteFile(

hPipe, // handle to pipe

ReplyBufferFull, // buffer to write from

datasize, // number of bytes to write

&datawritten, // number of bytes written

NULL); // not overlapped I/O

if (!fSuccess || datasize != datawritten)

{

printf("\nWrite to pipe failed.");

}

free(ReplyBufferFull);

continue;

}

vector<string> list = split(RequestBuffer, ':');

char req[10];

strcpy(req, list.at(0).c\_str());

printf("\nreq = %s", req);

if (stricmp(req, "uninstall") == 0)

{

printf("\nUNINSTALL request from UI");

ProcessUIAction\_Uninstall(RequestBuffer);

}

else if (stricmp(req, "install") == 0)

{

printf("\nINSTALL request from UI");

ProcessUIAction\_Install(RequestBuffer);

}

else if (stricmp(req, "copy") == 0)

{

printf("\nCopy request from UI");

ProcessUIAction\_Copy(RequestBuffer);

}

else if (stricmp(req, "getinfo") == 0)

{

printf("\nGETINFO request from UI");

ProcessUIAction\_GetInfo(RequestBuffer);

}

}

} // if (fconnected)

printf("\nClosing Pipe");

CloseHandle(hPipe);

} // while loop

return ERROR\_SUCCESS;

}

int ProcessUIAction\_Install(char \*RequestBuffer)

{

vector<string> list = split(RequestBuffer, ':');

PCLIENTINFO pClientInfo = GetClientRecord(list.at(1).c\_str());

pClientInfo->JobStatus = JOB\_STATUS\_NEW;

pClientInfo->Job.action = ACTION\_INSTALL;

strcpy(pClientInfo->Job.app, list.at(2).c\_str());

strcpy(pClientInfo->Job.version, list.at(3).c\_str());

pClientInfo->Job.action\_status = ACTION\_STATUS\_NEW;

return RSMC\_SUCCESS;

}

int ProcessUIAction\_Uninstall(char \*RequestBuffer)

{

vector<string> list = split(RequestBuffer, ':');

PCLIENTINFO pClientInfo = GetClientRecord(list.at(1).c\_str());

pClientInfo->JobStatus = JOB\_STATUS\_NEW;

pClientInfo->Job.action = ACTION\_UNINSTALL;

strcpy(pClientInfo->Job.app, list.at(2).c\_str());

strcpy(pClientInfo->Job.version, list.at(3).c\_str());

pClientInfo->Job.action\_status = ACTION\_STATUS\_NEW;

return RSMC\_SUCCESS;

}

int ProcessUIAction\_Copy(char \*RequestBuffer)

{

vector<string> list = split(RequestBuffer, ':');

PCLIENTINFO pClientInfo = GetClientRecord(list.at(1).c\_str());

pClientInfo->JobStatus = JOB\_STATUS\_NEW;

pClientInfo->Job.action = ACTION\_COPY;

strcpy(pClientInfo->Job.app, list.at(2).c\_str());

pClientInfo->Job.action\_status = ACTION\_STATUS\_NEW;

return RSMC\_SUCCESS;

}

int ProcessUIAction\_GetInfo(char \*RequestBuffer)

{

vector<string> list = split(RequestBuffer, ':');

PCLIENTINFO pClientInfo = GetClientRecord(list.at(1).c\_str());

pClientInfo->JobStatus = JOB\_STATUS\_NEW;

pClientInfo->Job.action = ACTION\_GETINFO;

strcpy(pClientInfo->Job.subaction, list.at(2).c\_str());

pClientInfo->Job.action\_status = ACTION\_STATUS\_NEW;

return RSMC\_SUCCESS;

}

SServiceExe.cpp

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <Windows.h>

#include <tchar.h>

#include <stdio.h>

#include "ServerCommon.h"

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

SERVICE\_STATUS g\_ServiceStatus = { 0 };

SERVICE\_STATUS\_HANDLE g\_StatusHandle = NULL;

HANDLE g\_ServiceStopEvent = INVALID\_HANDLE\_VALUE;

VOID WINAPI ServiceMain(DWORD argc, LPTSTR \*argv);

VOID WINAPI ServiceCtrlHandler(DWORD);

#define SERVICE\_NAME \_T("RSMC Server Service")

int \_tmain(int argc, TCHAR \*argv[])

{

DWORD Status = E\_FAIL;

// initiate the use of winsock library

WSAData wsaData; // initialize

int Result = WSAStartup(MAKEWORD(2, 1), &wsaData);

if (Result != NO\_ERROR) // check for errors

{

OutputDebugString(\_T("Error at WSAStartup()"));

goto EXIT;

}

//strcpy()

// Start a thread that will perform the main task of the service

// create threat for managing communication with client and sending jobs to client

HANDLE hThreadCommJob = CreateThread(NULL, 0, ServiceWorkerThread\_ManageCommunicationAndJob, NULL, 0, NULL);

// create thread for managing job status from clients

HANDLE hThreadJobStatus = CreateThread(NULL, 0, ServiceWorkerThread\_ManageJostStatus, NULL, 0, NULL);

HANDLE hThreadUI = CreateThread(NULL, 0, ServiceWorkerThread\_UI, NULL, 0, NULL);

HANDLE hThread[] = { hThreadCommJob, hThreadJobStatus, hThreadUI };

// Wait until our worker thread exits signaling that the service needs to stop

WaitForMultipleObjects(2, hThread, TRUE, INFINITE);

EXIT:

return 0;

}

Service.cpp

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <Windows.h>

#include <tchar.h>

#include <stdio.h>

#include "ServerCommon.h"

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

SERVICE\_STATUS g\_ServiceStatus = { 0 };

SERVICE\_STATUS\_HANDLE g\_StatusHandle = NULL;

HANDLE g\_ServiceStopEvent = INVALID\_HANDLE\_VALUE;

VOID WINAPI ServiceMain(DWORD argc, LPTSTR \*argv);

VOID WINAPI ServiceCtrlHandler(DWORD);

DWORD WINAPI ServiceWorkerThread\_ManageJostStatus(LPVOID lpParam);

DWORD WINAPI ServiceWorkerThread\_ManageCommunicationAndJob(LPVOID lpParam);

VOID WINAPI ServiceCtrlHandler(DWORD CtrlCode);

#define SERVICE\_NAME \_T("RSMC Server Service")

int \_tmain(int argc, TCHAR \*argv[])

{

SERVICE\_TABLE\_ENTRY ServiceTable[] =

{

{ SERVICE\_NAME, (LPSERVICE\_MAIN\_FUNCTION)ServiceMain },

{ NULL, NULL }

};

if (StartServiceCtrlDispatcher(ServiceTable) == FALSE)

{

return GetLastError();

}

return 0;

}

VOID WINAPI ServiceMain(DWORD argc, LPTSTR \*argv)

{

DWORD Status = E\_FAIL;

// Register our service control handler with the SCM

g\_StatusHandle = RegisterServiceCtrlHandler(SERVICE\_NAME, ServiceCtrlHandler);

if (g\_StatusHandle == NULL)

{

goto EXIT;

}

// Tell the service controller we are starting

ZeroMemory(&g\_ServiceStatus, sizeof(g\_ServiceStatus));

g\_ServiceStatus.dwServiceType = SERVICE\_WIN32\_OWN\_PROCESS;

g\_ServiceStatus.dwControlsAccepted = 0;

g\_ServiceStatus.dwCurrentState = SERVICE\_START\_PENDING;

g\_ServiceStatus.dwWin32ExitCode = 0;

g\_ServiceStatus.dwServiceSpecificExitCode = 0;

g\_ServiceStatus.dwCheckPoint = 0;

if (SetServiceStatus(g\_StatusHandle, &g\_ServiceStatus) == FALSE)

{

OutputDebugString(\_T(

"RSMC Server Service: ServiceMain: SetServiceStatus returned error"));

}

/\*

\* Perform tasks necessary to start the service here

\*/

// Create a service stop event to wait on later

g\_ServiceStopEvent = CreateEvent(NULL, TRUE, FALSE, NULL);

if (g\_ServiceStopEvent == NULL)

{

// Error creating event

// Tell service controller we are stopped and exit

g\_ServiceStatus.dwControlsAccepted = 0;

g\_ServiceStatus.dwCurrentState = SERVICE\_STOPPED;

g\_ServiceStatus.dwWin32ExitCode = GetLastError();

g\_ServiceStatus.dwCheckPoint = 1;

if (SetServiceStatus(g\_StatusHandle, &g\_ServiceStatus) == FALSE)

{

OutputDebugString(\_T(

"RSMC Server Service: ServiceMain: SetServiceStatus returned error"));

}

goto EXIT;

}

// Tell the service controller we are started

g\_ServiceStatus.dwControlsAccepted = SERVICE\_ACCEPT\_STOP;

g\_ServiceStatus.dwCurrentState = SERVICE\_RUNNING;

g\_ServiceStatus.dwWin32ExitCode = 0;

g\_ServiceStatus.dwCheckPoint = 0;

if (SetServiceStatus(g\_StatusHandle, &g\_ServiceStatus) == FALSE)

{

OutputDebugString(\_T(

"RSMC Server Service: ServiceMain: SetServiceStatus returned error"));

}

// initiate the use of winsock library

WSAData wsaData; // initialize

int Result = WSAStartup(MAKEWORD(2, 1), &wsaData);

if (Result != NO\_ERROR) // check for errors

{

OutputDebugString(\_T("Error at WSAStartup()"));

goto EXIT;

}

// Start a thread that will perform the main task of the service

// create thread for managing job status from clients

HANDLE hThreadJobStatus = CreateThread(NULL, 0, ServiceWorkerThread\_ManageJostStatus, NULL, 0, NULL);

// create threat for managing communication with client and sending jobs to client

HANDLE hThreadCommJob = CreateThread(NULL, 0, ServiceWorkerThread\_ManageCommunicationAndJob, NULL, 0, NULL);

HANDLE hThread[] = { hThreadJobStatus, hThreadCommJob };

// Wait until our worker thread exits signaling that the service needs to stop

WaitForMultipleObjects(2, hThread, TRUE, INFINITE);

/\*

\* Perform any cleanup tasks

\*/

CloseHandle(g\_ServiceStopEvent);

// Tell the service controller we are stopped

g\_ServiceStatus.dwControlsAccepted = 0;

g\_ServiceStatus.dwCurrentState = SERVICE\_STOPPED;

g\_ServiceStatus.dwWin32ExitCode = 0;

g\_ServiceStatus.dwCheckPoint = 3;

if (SetServiceStatus(g\_StatusHandle, &g\_ServiceStatus) == FALSE)

{

OutputDebugString(\_T(

"RSMC Server Service: ServiceMain: SetServiceStatus returned error"));

}

EXIT:

return;

}

VOID WINAPI ServiceCtrlHandler(DWORD CtrlCode)

{

switch (CtrlCode)

{

case SERVICE\_CONTROL\_STOP:

if (g\_ServiceStatus.dwCurrentState != SERVICE\_RUNNING)

break;

/\*

\* Perform tasks necessary to stop the service here

\*/

g\_ServiceStatus.dwControlsAccepted = 0;

g\_ServiceStatus.dwCurrentState = SERVICE\_STOP\_PENDING;

g\_ServiceStatus.dwWin32ExitCode = 0;

g\_ServiceStatus.dwCheckPoint = 4;

if (SetServiceStatus(g\_StatusHandle, &g\_ServiceStatus) == FALSE)

{

OutputDebugString(\_T(

"RMSC Server Service: ServiceCtrlHandler: SetServiceStatus returned error"));

}

// This will signal the worker thread to start shutting down

SetEvent(g\_ServiceStopEvent);

break;

default:

break;

}

}

1. Client Side Code

ClientCommon.h

#pragma once

#define \_WINSOCK\_DEPRECATED\_NO\_WARNINGS

#define \_CRT\_SECURE\_NO\_WARNNINGS

#include <Windows.h>

#include <iostream>

#include <string>

#include <stdio.h>

#include <ctime>

#include <sstream>

#include <vector>

#include <tchar.h>

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

using namespace std;

// action constants

#define ACTION\_INSTALL 1

#define ACTION\_UNINSTALL 2

#define ACTION\_COPY 3

#define ACTION\_GETINFO 4

#define ACTION\_EXIT 9

#define ACTION\_INVALID -9

#define RSMC\_ERROR 1

#define RSMC\_SUCCESS 0

#define MAX\_APP\_NAME 50

#define MAX\_VERSION\_STRING\_SIZE 30

#define MAX\_ACTION\_STRING 20

#define MAX\_RESPONSE\_STRING 512

struct \_JobInfo

{

int jobid;

int action;

char app[MAX\_APP\_NAME+1];

char version[MAX\_VERSION\_STRING\_SIZE+1];

char subaction[MAX\_ACTION\_STRING+1];

char subaction\_arg[MAX\_ACTION\_STRING+1];

char job\_string[512];

char \*GetInfo\_Output;

char action\_status\_string[MAX\_RESPONSE\_STRING+1];

int action\_status; // RSMC\_SUCCESS or RSMC\_ERROR

char response\_stringToBeSent[MAX\_RESPONSE\_STRING+1];

};

typedef struct \_JobInfo CMDINFO, JOB, \*PJOB;

struct \_ServerInfo

{

char ServerIP[16];

u\_short CommPort;

u\_short StatusPort;

};

typedef \_ServerInfo SERVERINFO, \*PSERVERINFO;

//function declearation

int Action\_PerformInstall(CMDINFO \* CmdInfo);

int Action\_PerformUnInstall(CMDINFO \* CmdInfo);

int Action\_PerformCopy(CMDINFO \*CmdInfo);

char \* Action\_PerformGetInfo(CMDINFO \*CmdInfo);

char \* GetWindowsService(void);

char \* GetWindowsProcesses(void);

char \* GetSoftwareInstalled(void);

char \* GetLogicalDrivesInfo();

int SendToServer(SOCKET s, char \*response);

// common functions - helper

vector<string> split(const string &s, char delim);

DWORD WINAPI ServiceWorkerThread\_ProcessServerRequest(LPVOID lpParam);

VOID WINAPI ServiceCtrlHandler(DWORD CtrlCode);

int GetIPAddress(SOCKET s, char \*IP);

int GetHostName(char \*Hostname);

void print(\_TCHAR \*msg);

int ConectToServerGetWaitingJob(PJOB Job);

int ProcessJob(PJOB Job);

int ConnectToServerToGiveJobStatus(PJOB Job);

void PrepareResponseString(PJOB Job);

void FreeJobMemory(PJOB Job);

ClientCommon.h

#define \_WINSOCK\_DEPRECATED\_NO\_WARNINGS

#define \_CRT\_SECURE\_NO\_WARNNINGS

#include <Windows.h>

#include <tchar.h>

#include <stdio.h>

#include "ClientCommon.h"

#ifdef CSERVICE\_RSMC

extern HANDLE g\_ServiceStopEvent;

#endif

void print(\_TCHAR \*msg)

{

#ifdef CSERVICE\_RSMC

OutputDebugString(msg);

#else

\_tprintf(msg);

#endif

}

SERVERINFO ServerInfo = {

"127.0.0.1",

55555,

55556

};

int registered\_flag = 0;

int GetIPAddress(SOCKET s, char \*IP)

{

sockaddr\_in SockAddr;

int addrlen = sizeof(SockAddr);

if (getsockname(s, (LPSOCKADDR)&SockAddr, &addrlen) == SOCKET\_ERROR)

{

return WSAGetLastError();

}

strcpy(IP, inet\_ntoa(SockAddr.sin\_addr));

return RSMC\_SUCCESS;

}

LSTATUS ReadRegistry(LPCSTR sPath, LPCSTR sKey, LPCSTR pBuffer, LPDWORD pBufferSize)

{

HKEY hKey;

LSTATUS nResult = ::RegOpenKeyExA(HKEY\_LOCAL\_MACHINE, sPath,

0, KEY\_READ | KEY\_WOW64\_64KEY, &hKey);

if (nResult == ERROR\_SUCCESS)

{

nResult = ::RegQueryValueExA(hKey, sKey, NULL, NULL,

(LPBYTE)pBuffer, pBufferSize);

RegCloseKey(hKey);

}

return (nResult);

}

int GetWinVersion(char \*osname, char \*osverion)

{

DWORD size = 100;

LSTATUS nResult = ReadRegistry("SOFTWARE\\Microsoft\\Windows NT\\CurrentVersion",

"ProductName", osname, &size);

nResult = ReadRegistry("SOFTWARE\\Microsoft\\Windows NT\\CurrentVersion",

"ReleaseId", osverion, &size);

return RSMC\_SUCCESS;

}

int GetHostName(char \*hostname)

{

int retval = GetEnvironmentVariableA("COMPUTERNAME", hostname, 100);

return RSMC\_SUCCESS;

}

// connects to sever, looks for waiting job, if job found, it creates the job object

// with detail and returns to caller 0

// if there is no job, it return 1

// closes the connection

//

int ConectToServerGetWaitingJob(PJOB Job)

{

// create socket

print(\_T("\nCreate socket ..."));

SOCKET sockSocket = socket(AF\_INET, SOCK\_STREAM, IPPROTO\_TCP); // create a socket

if (sockSocket == INVALID\_SOCKET) // check for errors

{

//char buff[500];

//sprintf(buff, "Error Creating socket, Error: %d", WSAGetLastError());

print(\_T("Error connecting to socket"));

return RSMC\_ERROR;

}

int err = 0;

//GetIPAddress(sockSocket, SelfIP, &err);

// connect to socket

struct sockaddr\_in ServerAddress;

ServerAddress.sin\_addr.s\_addr = inet\_addr(ServerInfo.ServerIP); // connect to the ipnuted IP

ServerAddress.sin\_family = AF\_INET;

ServerAddress.sin\_port = htons(ServerInfo.CommPort); ///(55555);

print(\_T("Connect to server socket ..."));

if (connect(sockSocket, (SOCKADDR\*)&ServerAddress, sizeof(ServerAddress)) == SOCKET\_ERROR) // check for errors

{

//char buff[500];

//sprintf(buff, "Error Creating socket, Error: %d", WSAGetLastError());

print(\_T("Failed to connect to server"));

closesocket(sockSocket);

return RSMC\_ERROR;

}

char request\_string[50] = "NewJob?";

print(\_T("\nsend reqest to server to check if pending job are available "));

if (SendToServer(sockSocket, request\_string) == RSMC\_ERROR)

{

//char buff[500];

//sprintf(buff, "Error Creating socket, Error: %d", WSAGetLastError());

print(\_T("Failed to send request"));

closesocket(sockSocket);

return RSMC\_ERROR;

}

char buffer[512];

int result;

do

{

print(\_T("\nrececive pending job answer from server"));

result = recv(sockSocket, buffer, 512, 0);

if (result <= 0)

break;

buffer[result] = '\0';

break;

} while (true);

if (result > 0)

{

strcpy(Job->job\_string, buffer);

}

// update the job object

closesocket(sockSocket);

return RSMC\_SUCCESS;

}

void PrepareResponseString(PJOB Job)

{

//jobid:action:status:action\_status:action\_status\_string

// if action is getinfo then we should send getinfo command output separately

sprintf(Job->response\_stringToBeSent, "%d:%d:%d:%s", Job->jobid, Job->action, Job->action\_status, Job->action\_status\_string);

}

int ConnectToServerToGiveJobStatus(PJOB Job)

{

// create socket

print(\_T("Create Socket ..."));

SOCKET sockSocket = socket(AF\_INET, SOCK\_STREAM, IPPROTO\_TCP); // create a socket

if (sockSocket == INVALID\_SOCKET) // check for errors

{

//char buff[500];

//sprintf(buff, "Error Creating socket, Error: %d", WSAGetLastError());

print(\_T("Error connecting to socket"));

return RSMC\_ERROR;

}

// connect to socket

struct sockaddr\_in ServerAddress;

ServerAddress.sin\_addr.s\_addr = inet\_addr(ServerInfo.ServerIP); // connect to the ipnuted IP

ServerAddress.sin\_family = AF\_INET;

ServerAddress.sin\_port = htons(ServerInfo.StatusPort); // should the port also be the argument?

print(\_T("\n Connect to server ..."));

if (connect(sockSocket, (SOCKADDR\*)&ServerAddress, sizeof(ServerAddress)) == SOCKET\_ERROR) // check for errors

{

//char buff[500];

//sprintf(buff, "Error Creating socket, Error: %d", WSAGetLastError());

print(\_T("Failed to connect to server"));

closesocket(sockSocket);

return RSMC\_ERROR;

}

PrepareResponseString(Job);

print(\_T("\nsend status update to sever..."));

printf("\n response\_string: %s", Job->response\_stringToBeSent);

if(SendToServer(sockSocket, Job->response\_stringToBeSent) == RSMC\_ERROR)

{

//char buff[500];

//sprintf(buff, "Error Creating socket, Error: %d", WSAGetLastError());

print(\_T("Failed to send request"));

closesocket(sockSocket);

return RSMC\_ERROR;

}

if (Job->action == ACTION\_GETINFO)

{

// send the get\_info\_output

if (SendToServer(sockSocket, Job->GetInfo\_Output) == RSMC\_ERROR)

{

print(\_T("Failed to send getinfo\_output"));

closesocket(sockSocket);

return RSMC\_ERROR;

}

}

// receive response on update

char buffer[512];

int result;

do

{

print(\_T("\nrececive pending job answer from server"));

result = recv(sockSocket, buffer, 512, 0);

if (result <= 0)

break;

buffer[result] = '\0';

break;

} while (true);

if (result > 0)

{

printf("\nreponse received: %s", buffer);

if (stricmp(buffer, "GetInfoDataMissing") == 0)

{

// GetInfo output was not received by server, handle it here

}

else if (stricmp(buffer, "ClientNotFouond") == 0)

{

// Client may not registered, client information not found in server

}

else if (stricmp(buffer, "JobNotFound") == 0)

{

// Jobid not found in the server for which the response was sent

}

else if (stricmp(buffer, "ThankYou") == 0)

{

// Success fully status received and updated.

}

}

// update the job object

print(\_T("Out of ConnectToServerToGiveJobStatus() - closing connection to server"));

closesocket(sockSocket);

return RSMC\_SUCCESS;

}

void FreeJobMemory(PJOB Job)

{

free(Job->GetInfo\_Output);

}

int RegisterWithServer()

{

// create socket

print(\_T("\nCreate socket ..."));

SOCKET sockSocket = socket(AF\_INET, SOCK\_STREAM, IPPROTO\_TCP); // create a socket

if (sockSocket == INVALID\_SOCKET) // check for errors

{

//char buff[500];

//sprintf(buff, "Error Creating socket, Error: %d", WSAGetLastError());

print(\_T("\nError connecting to socket"));

return RSMC\_ERROR;

}

// connect to socket

struct sockaddr\_in ServerAddress;

ServerAddress.sin\_addr.s\_addr = inet\_addr(ServerInfo.ServerIP); // connect to the ipnuted IP

ServerAddress.sin\_family = AF\_INET;

ServerAddress.sin\_port = htons(ServerInfo.CommPort); ///(55555);

printf("\nConnecting to server socket %s...", ServerInfo.ServerIP);

if (connect(sockSocket, (SOCKADDR\*)&ServerAddress, sizeof(ServerAddress)) == SOCKET\_ERROR) // check for errors

{

//char buff[500];

//sprintf(buff, "Error Creating socket, Error: %d", WSAGetLastError());

printf("\nFailed to connect to server: error - %d", WSAGetLastError());

closesocket(sockSocket);

return RSMC\_ERROR;

}

char request\_string[100] = "Register?:Window 10:1803";

char hostname[100], osname[100], version[100];

GetHostName(hostname);

GetWinVersion(osname, version);

sprintf(request\_string, "Register?:%s:%s:%s", hostname, osname, version);

print(\_T("\nRegistration request send. "));

if (SendToServer(sockSocket, request\_string) == RSMC\_ERROR)

{

//char buff[500];

//sprintf(buff, "Error Creating socket, Error: %d", WSAGetLastError());

print(\_T("Failed to send request"));

closesocket(sockSocket);

return RSMC\_ERROR;

}

char buffer[512];

int result;

do

{

print(\_T("\nRegistration request response:"));

result = recv(sockSocket, buffer, 512, 0);

if (result <= 0)

break;

buffer[result] = '\0';

break;

} while (true);

if (result > 0)

{

printf("%s", buffer);

if (stricmp(buffer, "registered") == 0)

{

print(\_T("\nRegistration Successful"));

return RSMC\_SUCCESS;

}

else

{

printf("\nregistrataion status: %s", buffer);

print(\_T("\nRegistration failed"));

}

}

return RSMC\_ERROR;

}

DWORD WINAPI ServiceWorkerThread\_ProcessServerRequest(LPVOID lpParam)

{

// register the client first

printf("\n=>ServiceWorkerThread\_ProcessServerRequest()");

do

{

printf("\n=>RegisterWithServer()");

int retval = RegisterWithServer();

printf("\n<=RegisterWithServer()");

if (retval == RSMC\_SUCCESS)

{

registered\_flag = 1;

break;

}

} while (registered\_flag == 0);

print(\_T("\nregistered with server successfully."));

print(\_T("\nStarting ProcessServerRequest Thread ..."));

#ifdef CSERVICE\_RSMC

// Periodically check if the service has been requested to stop

while (WaitForSingleObject(g\_ServiceStopEvent, 0) != WAIT\_OBJECT\_0)

#else

while (true)

#endif

{

printf("\nWaiting for next poll time to elapse ...");

Sleep(60 \* 1000);

JOB Job;

// connect to server and check if there is any job

print(\_T("\nConnecting to server for waiting job ..."));

printf("\n=>ConectToServerGetWaitingJob()");

int status = ConectToServerGetWaitingJob(&Job);

printf("\n<=ConectToServerGetWaitingJob()");

// if job is found process it

if (status == RSMC\_SUCCESS)

{

printf("\n Response for GetWaitingJob -> %s", Job.job\_string);

if ((stricmp(Job.job\_string, "UnkownRequest") == 0) ||

(stricmp(Job.job\_string, "NoNewJob") == 0))

continue;

else

{

printf("\n=>ProcessJob()");

ProcessJob(&Job);

printf("\n<=ProcessJob()");

}

}

// send the result/status of the job requested by server

print(\_T("\nConnecting to server for updating job status and result ..."));

status = ConnectToServerToGiveJobStatus(&Job);

print(\_T("Done."));

FreeJobMemory(&Job);

print(\_T("ServiceWorkerThread\_ManageJostStatus"));

}

return RSMC\_SUCCESS;

}

Copy.cpp

#include "ClientCommon.h"

int Action\_PerformCopy(CMDINFO \*CmdInfo)

{

char \*cmd = "mount\_repos.bat";

char \*repos\_copyfolder = "p:\\copyfolder";

char \*destination\_folder = ".\\copiedfiles";

// mount the repository drive

system(cmd);

// copy the file

char cmd\_copy[512];

sprintf(cmd\_copy, "copy %s\\%s %s /y", repos\_copyfolder, CmdInfo->app, destination\_folder);

printf("\nCopying %s from repository...", CmdInfo->app);

int retval = system(cmd\_copy);

if (retval == ERROR)

{

printf("ERROR: %d", retval);

return ERROR;

}

printf("Done.\n");

return RSMC\_SUCCESS;

}

GetInfo.cpp

#include "ClientCommon.h"

char \* Action\_PerformGetInfo(CMDINFO \*CmdInfo)

{

char \*message;

if (stricmp(CmdInfo->subaction, "Service") == 0)

{

message = GetWindowsService();

if ( message == NULL)

return NULL;

}

else if (stricmp(CmdInfo->subaction, "Process") == 0)

{

message = GetWindowsProcesses();

if (message == NULL)

return NULL;

}

else if (stricmp(CmdInfo->subaction, "Softwares") == 0)

{

message = GetSoftwareInstalled();

if (message == NULL)

return NULL;

}

else if (stricmp(CmdInfo->subaction, "Disk") == 0)

{

message = GetLogicalDrivesInfo();

if (message == NULL)

return NULL;

}

return message;

}

unsigned long GetFileSize(char \*filename)

{

FILE \* f = fopen(filename, "r");

fseek(f, 0, SEEK\_END);

unsigned long len = (unsigned long)ftell(f);

fclose(f);

return len;

}

char \* GetLogicalDrivesInfo()

{

//char \*cmd = "wmic /output:disk.out logicaldisk get size, freespace, caption";

char \*cmd = "powershell -executionpolicy bypass -command \"Get-WMIObject Win32\_LogicalDisk\" > disk.out";

system(cmd);

long size = GetFileSize("disk.out");

char \*output = (char\*)malloc(size);

if (output == NULL)

return NULL;

FILE\* fp = fopen("disk.out", "r");

if (fp == NULL)

{

printf("ERROR: Failed to read disk.out!");

return NULL;

}

int s = fread((void\*)output, 1, size, fp);

if (size >= s)

return output;

else

{

free(output);

return NULL;

}

}

char \* GetSoftwareInstalled()

{

char \*cmd = "powershell -executionpolicy ByPass -Command \"Get-ItemProperty HKLM:\\Software\\Wow6432Node\\Microsoft\\Windows\\CurrentVersion\\Uninstall\\\* | Select-Object DisplayName, DisplayVersion, Publisher, InstallDate\" > SoftwareList.out";

system(cmd);

long size = GetFileSize("SoftwareList.out");

char \*output = (char\*)malloc(size);

if (output == NULL)

return NULL;

FILE\* fp = fopen("SoftwareList.out", "rb");

if (fp == NULL)

{

printf("ERROR: Failed to read SoftwareList.out!");

return NULL;

}

int s = fread((void\*)output, 1, size, fp);

if (s == size)

return output;

else

{

free(output);

return NULL;

}

}

char \* GetWindowsService()

{

char \*cmd = "powershell -executionpolicy ByPass -Command Get-Service > GetService.out";

system(cmd);

long size = GetFileSize("GetService.out");

char \*output = (char\*)malloc(size);

if (output == NULL)

return NULL;

FILE\* fp = fopen("GetService.out", "rb");

if (fp == NULL)

{

printf("ERROR: Failed to read GetService.out!");

return NULL;

}

int s = fread((void\*)output, 1, size, fp);

if (s == size)

return output;

else

{

free(output);

return NULL;

}

}

char \* GetWindowsProcesses()

{

char \*output;

char \*cmd = "tasklist > tasklist.out";

system(cmd);

long size = GetFileSize("tasklist.out");

output = (char\*)malloc(size);

if (output == NULL)

return NULL;

FILE\* fp = fopen("tasklist.out", "rb");

if (fp == NULL)

{

printf("ERROR: Failed to read tasklist.out!");

return NULL;

}

int s = fread((void\*)output, 1, size, fp);

if (s == size)

return output;

else

{

free(output);

return NULL;

}

return RSMC\_SUCCESS;

}

Install.cpp

#include "clientcommon.h"

int Action\_PerformInstall(CMDINFO \* CmdInfo)

{

char \*cmd = "mount\_repos.bat";

system(cmd);

char cmd\_install[100];

sprintf(cmd\_install, "p:\\%s\\%s\\%s-install.bat", CmdInfo->app, CmdInfo->version, CmdInfo->app);

printf("\nInstalling ... Cmd:[%s]", cmd\_install);

int retval = system(cmd\_install);

if (retval == RSMC\_ERROR)

{

printf("ERROR: %d", retval);

return RSMC\_ERROR;

}

printf("Done.\n");

return RSMC\_SUCCESS;

}

int Action\_PerformUnInstall(CMDINFO \* CmdInfo)

{

char \*cmd = "mount\_repos.bat";

system(cmd);

char cmd\_uninstall[100];

sprintf(cmd\_uninstall, "p:\\%s\\%s\\%s-uninstall.bat", CmdInfo->app, CmdInfo->version, CmdInfo->app);

printf("\nUnInstalling ... Cmd:[%s]", cmd\_uninstall);

int retval = system(cmd\_uninstall);

if (retval == RSMC\_ERROR)

{

printf("ERROR: %d", retval);

return RSMC\_ERROR;

}

printf("Done.\n");

return RSMC\_SUCCESS;

}

Client.cpp

#include "ClientCommon.h"

// splits a given string based on delimter provided

// e.g.: "1:appname:version" split into

// 1

// appname

// version

//

vector<string> split(const string &s, char delim) {

vector<string> result;

stringstream ss(s);

string item;

while (getline(ss, item, delim)) {

result.push\_back(item);

}

return result;

}

//

// Process the server request and extract action and relevant data for each action

// request may look like this:

// 1:appname:version

// 2:appname

// 3:filename

// 4:Service or 4:Process

// Input: ActionString

// Output: Fill the CmdInfo passed with extracted Action and its relevant data

// returns: SUCCESS on successful extraction else ERROR on failure

///

int GetActionDetails(char \*ActionString, CMDINFO\* CmdInfo)

{

vector<string> list = split(ActionString, ':');

int retval = RSMC\_SUCCESS;

CmdInfo->jobid = atoi(list.at(0).c\_str());

CmdInfo->action = atoi(list.at(1).c\_str());

printf("\n jobid: %d, action = %d", CmdInfo->jobid, CmdInfo->action);

switch (CmdInfo->action)

{

case ACTION\_INSTALL:

strcpy(CmdInfo->app, list.at(2).c\_str());

strcpy(CmdInfo->version, list.at(3).c\_str());

printf("\nApp: %s, version: %s", CmdInfo->app, CmdInfo->version);

break;

case ACTION\_UNINSTALL:

strcpy(CmdInfo->app, list.at(2).c\_str());

strcpy(CmdInfo->version, list.at(3).c\_str());

break;

case ACTION\_COPY:

strcpy(CmdInfo->app, list.at(2).c\_str());

break;

case ACTION\_GETINFO:

strcpy(CmdInfo->subaction, list.at(2).c\_str());

break;

default:

retval = RSMC\_ERROR;

break;

}

return retval;

}

int SendToServer(SOCKET s, char \*response)

{

std::string message = response;

print(\_T("\nSending response ..."));

if (send(s, message.c\_str(), message.length(), 0) == SOCKET\_ERROR)

{

//std::cout << "Error at send(): " << WSAGetLastError() << std::endl;

print(\_T("Errot sending message."));

return RSMC\_ERROR;

}

print(\_T("Done.\n"));

return RSMC\_SUCCESS;

}

int ProcessJob(PJOB Job)

{

char ActionString[512];

strcpy(ActionString, Job->job\_string);

int retval = RSMC\_SUCCESS;

char \*GetInfoOutput = NULL;

printf("\n ActionString = %s", ActionString);

retval = GetActionDetails(ActionString, Job);

if (retval == RSMC\_ERROR)

{

printf("\nError: Invalid Action sent by server!");

//SendResponseToServer(s, "ERROR: Invalid Request, Cannot process!");

Job->action\_status = ACTION\_INVALID;

return RSMC\_ERROR;

}

else

{

switch (Job->action)

{

case ACTION\_INSTALL:

retval = Action\_PerformInstall(Job);

if (retval == RSMC\_SUCCESS)

strcpy(Job->action\_status\_string, "SUCCESS - Installation Successfull!");

else

strcpy(Job->action\_status\_string, "ERROR - Installation failed!");

Job->action\_status = retval;

break;

case ACTION\_UNINSTALL:

retval = Action\_PerformUnInstall(Job);

if (retval == RSMC\_SUCCESS)

strcpy(Job->action\_status\_string, "SUCCESS - UnInstallation Successfull!");

else

strcpy(Job->action\_status\_string, "ERROR - UnInstallation failed!");

Job->action\_status = retval;

break;

case ACTION\_COPY:

retval = Action\_PerformCopy(Job);

if (retval == RSMC\_SUCCESS)

strcpy(Job->action\_status\_string, "SUCCESS - Copy Successfull!");

else

strcpy(Job->action\_status\_string, "ERROR - Copy failed!");

Job->action\_status = retval;

break;

case ACTION\_GETINFO:

char \*Info;

Info = Action\_PerformGetInfo(Job);

if (retval == RSMC\_SUCCESS)

{

strcpy(Job->action\_status\_string, "SUCCESS - GetInfo Successfull!");

Job->GetInfo\_Output = Info;

}

else

strcpy(Job->action\_status\_string, "ERROR - GetInfo failed!");

Job->action\_status = retval;

break;

default:

// we should never reach here,

// if reached this is a critical problem in the code

printf("\nERROR: Invalid Action requested.");

break;

}

}

return RSMC\_SUCCESS;

}

CServiceExe.cpp

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <Windows.h>

#include <tchar.h>

#include <stdio.h>

#include "ClientCommon.h"

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

extern SERVERINFO ServerInfo;

int main(int argc, char \*argv[])

{

DWORD Status = E\_FAIL;

if (argc < 2)

{

printf("\nRMSC Client Console\n");

printf("Usage: %s <RMSC Server IP>\n", argv[0]);

return 1;

}

strcpy(ServerInfo.ServerIP, argv[1]);

// initiate the use of winsock library

WSAData wsaData; // initialize

printf("RSMC Server IP provided: %s", ServerInfo.ServerIP);

print(\_T("\nStarting RSMC Client ..."));

print(\_T("\nRuning WSAStartup()..."));

int Result = WSAStartup(MAKEWORD(2, 1), &wsaData);

if (Result != NO\_ERROR) // check for errors

{

print(\_T("Error at WSAStartup()"));

return 1;

}

print(\_T("Done."));

// Start a thread that will perform the main task of the service

// create thread for managing job status from clients

HANDLE hThread = CreateThread(NULL, 0, ServiceWorkerThread\_ProcessServerRequest, NULL, 0, NULL);

// Wait until our worker thread exits signaling that the service needs to stop

WaitForSingleObject(hThread, INFINITE);

return 0;

}

CService.cpp

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <Windows.h>

#include <tchar.h>

#include <stdio.h>

#include "ClientCommon.h"

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

#define RSMC\_ERROR 1

#define RSMC\_SUCCESS 0

SERVICE\_STATUS g\_ServiceStatus = { 0 };

SERVICE\_STATUS\_HANDLE g\_StatusHandle = NULL;

HANDLE g\_ServiceStopEvent = INVALID\_HANDLE\_VALUE;

VOID WINAPI ServiceMain(DWORD argc, LPTSTR \*argv);

VOID WINAPI ServiceCtrlHandler(DWORD);

DWORD WINAPI ServiceWorkerThread\_ProcessServerRequest(LPVOID lpParam);

VOID WINAPI ServiceCtrlHandler(DWORD CtrlCode);

void GetIPAddress(SOCKET s, char \*IP, int \*err);

#define SERVICE\_NAME \_T("RSMC Client Service")

int \_tmain(int argc, TCHAR \*argv[])

{

SERVICE\_TABLE\_ENTRY ServiceTable[] =

{

{ SERVICE\_NAME, (LPSERVICE\_MAIN\_FUNCTION)ServiceMain },

{ NULL, NULL }

};

if (StartServiceCtrlDispatcher(ServiceTable) == FALSE)

{

return GetLastError();

}

return 0;

}

VOID WINAPI ServiceMain(DWORD argc, LPTSTR \*argv)

{

DWORD Status = E\_FAIL;

// Register our service control handler with the SCM

g\_StatusHandle = RegisterServiceCtrlHandler(SERVICE\_NAME, ServiceCtrlHandler);

if (g\_StatusHandle == NULL)

{

goto EXIT;

}

// Tell the service controller we are starting

ZeroMemory(&g\_ServiceStatus, sizeof(g\_ServiceStatus));

g\_ServiceStatus.dwServiceType = SERVICE\_WIN32\_OWN\_PROCESS;

g\_ServiceStatus.dwControlsAccepted = 0;

g\_ServiceStatus.dwCurrentState = SERVICE\_START\_PENDING;

g\_ServiceStatus.dwWin32ExitCode = 0;

g\_ServiceStatus.dwServiceSpecificExitCode = 0;

g\_ServiceStatus.dwCheckPoint = 0;

if (SetServiceStatus(g\_StatusHandle, &g\_ServiceStatus) == FALSE)

{

print(\_T("RSMC Client Service: ServiceMain: SetServiceStatus returned error"));

}

/\*

\* Perform tasks necessary to start the service here

\*/

// Create a service stop event to wait on later

g\_ServiceStopEvent = CreateEvent(NULL, TRUE, FALSE, NULL);

if (g\_ServiceStopEvent == NULL)

{

// Error creating event

// Tell service controller we are stopped and exit

g\_ServiceStatus.dwControlsAccepted = 0;

g\_ServiceStatus.dwCurrentState = SERVICE\_STOPPED;

g\_ServiceStatus.dwWin32ExitCode = GetLastError();

g\_ServiceStatus.dwCheckPoint = 1;

if (SetServiceStatus(g\_StatusHandle, &g\_ServiceStatus) == FALSE)

{

print(\_T("RSMC Client Service: ServiceMain: SetServiceStatus returned error"));

}

goto EXIT;

}

// Tell the service controller we are started

g\_ServiceStatus.dwControlsAccepted = SERVICE\_ACCEPT\_STOP;

g\_ServiceStatus.dwCurrentState = SERVICE\_RUNNING;

g\_ServiceStatus.dwWin32ExitCode = 0;

g\_ServiceStatus.dwCheckPoint = 0;

if (SetServiceStatus(g\_StatusHandle, &g\_ServiceStatus) == FALSE)

{

print(\_T("RSMC Client Service: ServiceMain: SetServiceStatus returned error"));

}

// initiate the use of winsock library

//initialize winsock and create a socket

WSAData wsaData; // initialize

int Result = WSAStartup(MAKEWORD(2, 1), &wsaData);

if (Result != NO\_ERROR) // check for errors

{

print(\_T("Error at WSAStartup()"));

goto EXIT;

}

// Start a thread that will perform the main task of the service

// create thread for managing job status from clients

HANDLE hThread = CreateThread(NULL, 0, ServiceWorkerThread\_ProcessServerRequest, NULL, 0, NULL);

// Wait until our worker thread exits signaling that the service needs to stop

WaitForSingleObject(hThread, INFINITE);

/\*

\* Perform any cleanup tasks

\*/

CloseHandle(g\_ServiceStopEvent);

// Tell the service controller we are stopped

g\_ServiceStatus.dwControlsAccepted = 0;

g\_ServiceStatus.dwCurrentState = SERVICE\_STOPPED;

g\_ServiceStatus.dwWin32ExitCode = 0;

g\_ServiceStatus.dwCheckPoint = 3;

if (SetServiceStatus(g\_StatusHandle, &g\_ServiceStatus) == FALSE)

{

print(\_T("RSMC Client Service: ServiceMain: SetServiceStatus returned error"));

}

EXIT:

return;

}

VOID WINAPI ServiceCtrlHandler(DWORD CtrlCode)

{

switch (CtrlCode)

{

case SERVICE\_CONTROL\_STOP:

if (g\_ServiceStatus.dwCurrentState != SERVICE\_RUNNING)

break;

/\*

\* Perform tasks necessary to stop the service here

\*/

g\_ServiceStatus.dwControlsAccepted = 0;

g\_ServiceStatus.dwCurrentState = SERVICE\_STOP\_PENDING;

g\_ServiceStatus.dwWin32ExitCode = 0;

g\_ServiceStatus.dwCheckPoint = 4;

if (SetServiceStatus(g\_StatusHandle, &g\_ServiceStatus) == FALSE)

{

print(\_T(

"RSMC Client Service: ServiceCtrlHandler: SetServiceStatus returned error"));

}

// This will signal the worker thread to start shutting down

SetEvent(g\_ServiceStopEvent);

break;

default:

break;

}

}