```
//
// Copyright (c) 2019 Phoenix Contact GmbH & Co. KG. All rights reserved.
// Licensed under the MIT. See LICENSE file in the project root for full license information.
// SPDX-License-Identifier: MIT
//
#include "Arp/System/ModuleLib/Module.h"
#include "Arp/System/Commons/Logging.h"
#include "Arp/Plc/AnsiC/Gds/DataLayout.h"
#include "Arp/Plc/AnsiC/Io/FbIoSystem.h"
#include "Arp/Plc/AnsiC/Io/Axio.h"
#include "Arp/System/Commons/Net/Socket.hpp"
#include "Arp/System/Commons/Runtime/Process.hpp"   // class Process
#include "Arp/System/Commons/Net/IpAddress.hpp"
#include "Arp/System/Commons/Threading/Thread.hpp"


#include <string>
#include <syslog.h>
#include <unistd.h>
#include <libgen.h>
#include <arpa/inet.h> // sockaddr_in
#include "sys/socket.h"
#include <iostream>
#include <vector>
#include <fstream>
#include <string>
#include <iostream>
#include <ctime>
#include <time.h>

using namespace std;
using namespace Arp;
using namespace Arp::System::Commons::Diagnostics::Logging;
using namespace Arp::System::Commons::Runtime;  // class Process
using namespace Arp::System::Commons::Net;
using namespace Arp::System::Commons::Threading;


class TcpInterfaceServer : private Loggable<TcpInterfaceServer> {
    public:
        TcpInterfaceServer();
        virtual ~TcpInterfaceServer();

    public: // Operations
        void Run(void *threadInfo);
        void Setup();

    private:
        string ProcessCommand(string &command);

    public:
        bool terminateServerThread;
        Socket listeningSocket;
        Thread serverThread;
};

inline TcpInterfaceServer::TcpInterfaceServer(void) :
    listeningSocket(SocketType::Tcp, SocketDomain::Ipv4, SocketBlockingMode::NoneBlocking),
    serverThread(this, &TcpInterfaceServer::Run)
{
}

inline void TcpInterfaceServer::Setup()
{
    // allow socket reuse
    listeningSocket.SetOptionReuseAddress(true);
    // control low priority socket server thread
```

```
    terminateServerThread = false;
    serverThread.SetAsynchronousCancelability(true);
    serverThread.SetPriority(1);
    serverThread.Start();
}


TcpInterfaceServer::~TcpInterfaceServer() {
        Log::Info("Terminate TcpInterfaceServer");
        // mark server thread as terminating
        terminateServerThread = true;
        // close existing sockets
        (void)listeningSocket.Shutdown();
        (void)listeningSocket.Close();
        try {
            if (serverThread.IsJoinable()) {
                serverThread.Join();
            }
            if (serverThread.IsRunning()) {
                serverThread.Terminate();
            }
            Log::Info("TcpInterfaceServer terminated");
        }
        catch (...) {
            Log::Error("Exception while shutting down");
        }
    }

    void TcpInterfaceServer::Run(void *threadInfo)
    {
        Log::Info("TcpInterfaceServer->Run: Starting socket");
        terminateServerThread = false;
        int port = 5555;

        // Bind the port to any local address.
        if (listeningSocket.Bind(0, port) != SocketError::None)
        {
            Log::Error("TcpInterfaceServer->Run: Unable to bind socket");
            return;
        }
        // Make socket a passive listener that processes incoming connection requests.
        if (listeningSocket.Listen(10) != SocketError::None)
        {
            Log::Error("TcpInterfaceServer->Run: Unable to listen");
            return;
        }
        //IpAddress remoteIp = IpAddress::Parse("127.0.0.1");
        //IpAddress remoteIp;
        //int remotePort;
        IpAddress remoteIp = IpAddress::Parse("192.168.1.10");
        int remotePort = 5555;
        SocketError error;
        // Receive any message and reply it directly to the sender.
        while (!terminateServerThread)
        {
            // Wait for the first client that requests a connection and accept it.
            Socket::Ptr newSocket = listeningSocket.Accept(remoteIp, remotePort, error);
            if (newSocket != nullptr)
            {
                Log::Info("Connected");
                string command = "";
                char buffer[256];
                int commandLength = 0;

                bool receivingData = true;
                while (receivingData)
                {
```

```
                // clean up buffer
                memset(buffer, 0, sizeof(buffer));
                // Receive a message from client

                int bytesReceived = newSocket->Receive(buffer, sizeof(buffer), error);

                command.append(buffer, bytesReceived);

                if (commandLength == 0)
                {
                    // Find first blank in telegram and resolve command length
                    int blankPos = command.find(" ");
                    string commandLengthString = command.substr(0, blankPos);
                    sscanf(commandLengthString.c_str(), "%d", &commandLength);
                    // Delete length information from command
                    command = command.substr(blankPos + 1, command.size());
                }
                if ((command.length() >= commandLength) || (bytesReceived == 0))
                {
                    receivingData = false;
                    Log::Info("TcpInterfaceServer->Run: receivingData = false");
                }
            }

            Log::Info("TcpInterfaceServer->Run: Command = '{0}'", command);

            string response = " " + ProcessCommand(command);
            response = to_string(response.size()) + response;
            newSocket->Send(response.c_str(), response.length(), error);

            Log::Info("TcpInterfaceServer->Run: Response = '{0}'", response);
            Thread::Sleep(250);
            (void)newSocket->Shutdown();
            (void)newSocket->Close();
        }
        // Prevent CPU blocking for non-blocking sockets
        Thread::Sleep(10);
    }
    Log::Info("serverThread is terminated");
}

string TcpInterfaceServer::ProcessCommand(string &command) {

    string response = "";

    // get command and generate response
    if (command.compare(0, 1, "A") == 0)
    {
        // Get response
        response = "B";
        Log::Info("TcpInterfaceServer->ProcessCommand: response = B");
    }
    else if (command.compare(0, 1, "E") == 0)
    {
        // terminate server by client
        terminateServerThread = true;
        Log::Info("TcpInterfaceServer->ProcessCommand: Terminate ServerThread requested
        by TCP Client");
    }
    else if (!listeningSocket.IsConnected())
    {
        // Error: client is disconnected
        response = "Error: client is disconnected";
    }
    return response;
}
```

```
TcpInterfaceServer* g_pTIS;

int main(int argc, char** argv)
{
   // Ask plcnext for access to its services
   // Use syslog for logging until the PLCnext logger is ready
   openlog ("tcp_socket", LOG_CONS | LOG_PID | LOG_NDELAY, LOG_LOCAL1);

   // Process command line arguments
   string acfSettingsRelPath("");

   if(argc != 2)
   {
      syslog (LOG_ERR, "Invalid command line arguments. Only relative path to the
      acf.settings file must be passed");
      return -1;
   }
   else
   {
      acfSettingsRelPath = argv[1];
      syslog(LOG_INFO, string("Arg Acf settings file path: " + acfSettingsRelPath).c_str());
   }

   char szExePath[PATH_MAX];
   ssize_t count = readlink("/proc/self/exe", szExePath, PATH_MAX);
   string strDirPath;
   if (count != -1) {
      strDirPath = dirname(szExePath);
   }
   string strSettingsFile(strDirPath);
      strSettingsFile += "/" + acfSettingsRelPath;
   syslog(LOG_INFO, string("Acf settings file path: " + strSettingsFile).c_str());

   // Intialize PLCnext module application
   // Arguments:
   //  arpBinaryDir:    Path to Arp binaries
   //  applicationName: Arbitrary Name of Application
   //  acfSettingsPath: Path to *.acf.settings document to set application up
   if (ArpSystemModule_Load("/usr/lib", "TCP_Server", strSettingsFile.c_str()) != 0)
   {
      syslog (LOG_ERR, "Could not load Arp System Module");
      return -1;
   }
   syslog (LOG_INFO, "Loaded Arp System Module");
   closelog();

   Log::Info("Start TcpInterfaceServer");
   g_pTIS = new TcpInterfaceServer();
   g_pTIS->Setup();
   sleep(30);

   while (true)
   {
     // Wait a short time before processing
      sleep(1);
   }
}
```