User Manual

# About this application

The goal of this exercise is to create a P2P messaging service, here is a little description about this answer:

This answer is a multi-purpose program which act as a client and a server, meaning there is no given role, each instance of the program will be able to connect to others, send messages to others (and receive from) and dispatch connection information to instances which need it.

Client and server roles will still be used but only as a vocabulary, to help identify each instance, since all instances can do everything.

The protocol used is based on the following messages:

* An identification message, used when a client want to connect to another one and when an instance ask for an unknown peer;
* A “message” message, to be able to work as a messaging service;
* A “message acknowledged” message, to verify if a message has been received;
* A peer request message, in case we want to send a message to a locally unknown peer;

UDP is used as communications protocol, connection-less protocol, it’s simple to use, but has some downsides:

* Connection-less protocol, so we need to think of a way to be able to reply back at a message;
* No reliable delivery of messages, i.e. messages can be lost;
* No integrity, i.e. messages might be corrupted.

This answer application has not been built to reply these problems, but the protocol used is helping to counter some problems:

* All messages contain the name of the sender, so if a peer already knows the connection data of the sender it can replay back;
* The identification message contains the name, the IP and the port of a peer, so if another peer receives an identification message, it has all it’s needed to send message to that peer.

# General presentation of the user interface

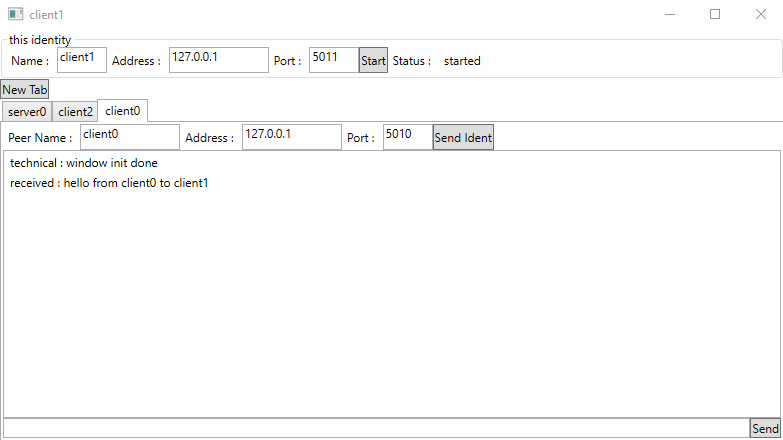
The application is written in C# with WFP under Visual Studio 2019 on a Windows 10 desktop.

It restricts quite a lot the compatibility but it helps (me at least) a lot:

* It allows me to create one UI component to send messages and one to receive these, meaning there is to tangled output like it often happen in command line program where input and output are written on screen at the same time;
* Basic functionalities of an application (message pump, lifecycle, etc.) are already written.

No third party is used, so it should work out of the box on a vanilla installation of VS2019;

So open the solution, build and launch to have a single instance of the program.



There are 2 main sections in this application:

* “this identity” panel;
* Tab controller which contains discussions initiated with peers.

The “this identity” panel contains mandatory information about the current instance:

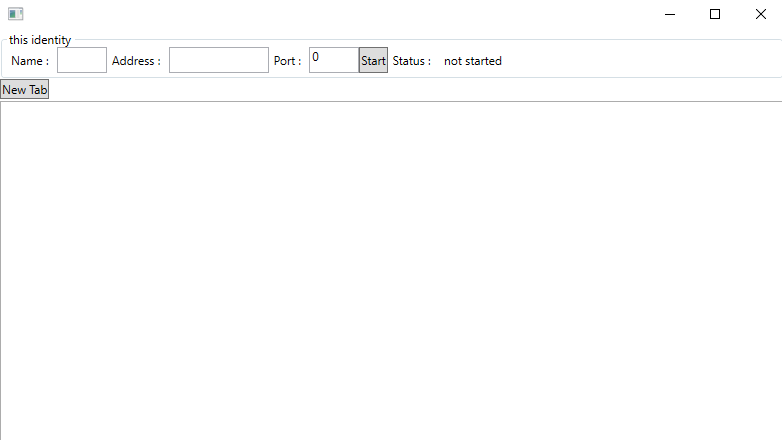
* The name used as a message sender and used by the other peers to contact “us”;
* Address and Port, which will be used by other peers to contact “us”, so we will open an UDP socket to listen on this Port;
* A Start button and a status message to start/have the status of the listening background thread.

The rest of the world:

* A “New Tab” button to initiate a conversation with another peer;
* A list of tabs containing already opened communications.

# How to use

You start with a window like this:



First things first, you should complete the “this identity” panel, for this example, we are “client0”, our address is “127.0.0.1” and we will listen on port 5010:



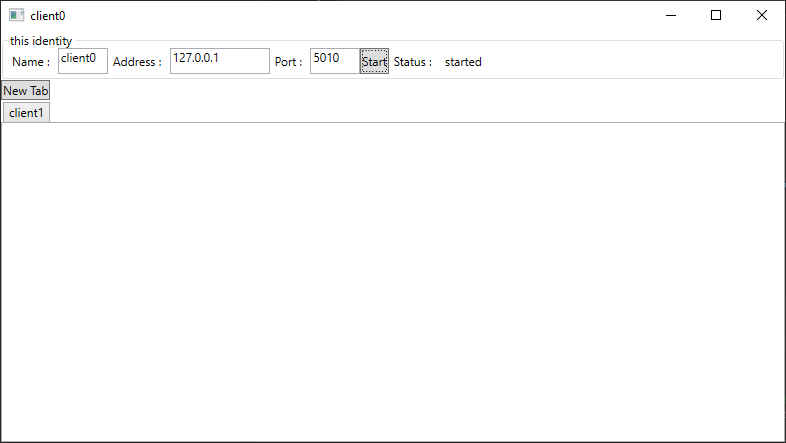
Next we click on the “Start” button, it should launch the background thread which will listen for input datagram:



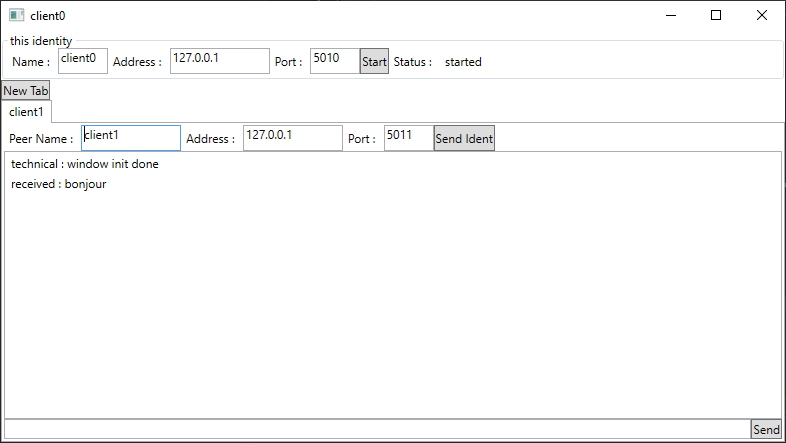
Now we have 3 cases:

* Someone else send a message;
* We know the connection data of the peer;
* We know only the name of the peer.

## First case: someone else send a message



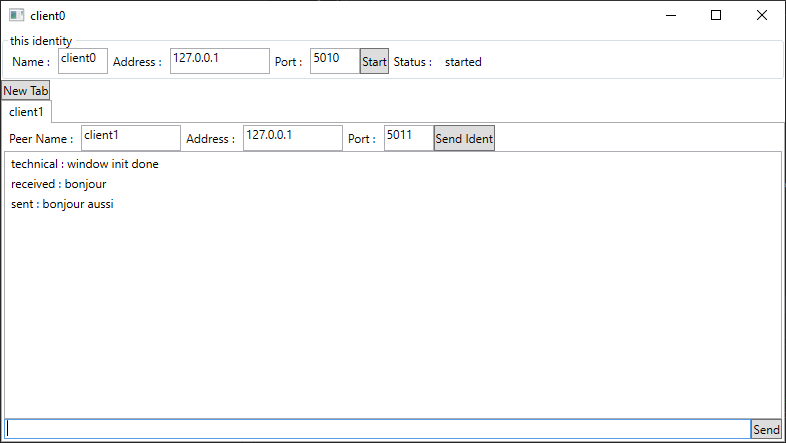
We can see that a tab named “client1” has been created but not opened …



… once we click on it, we see all information sent by “cient1” like it’s connection information, and a first message with the text “bonjour”.

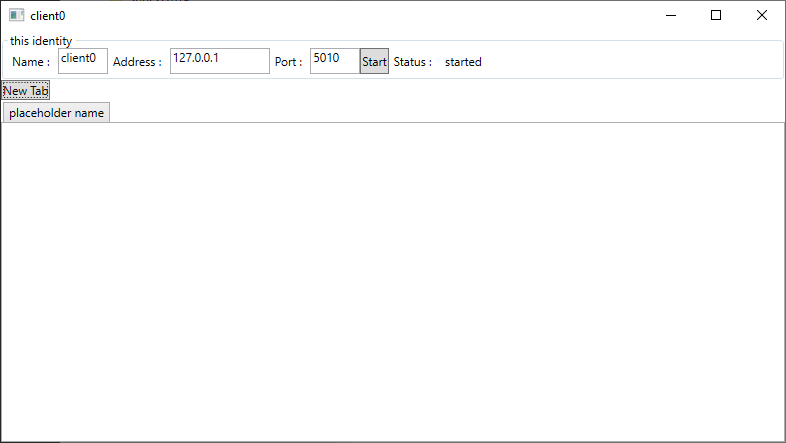
The message which begin with “technical” is just here to help debugging, since this is an exercise, I choose to let them visible.

Now we can reply to “client1” without worry.

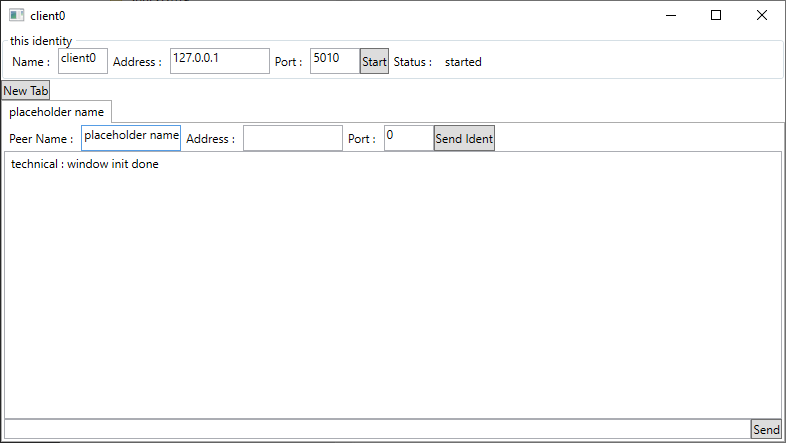


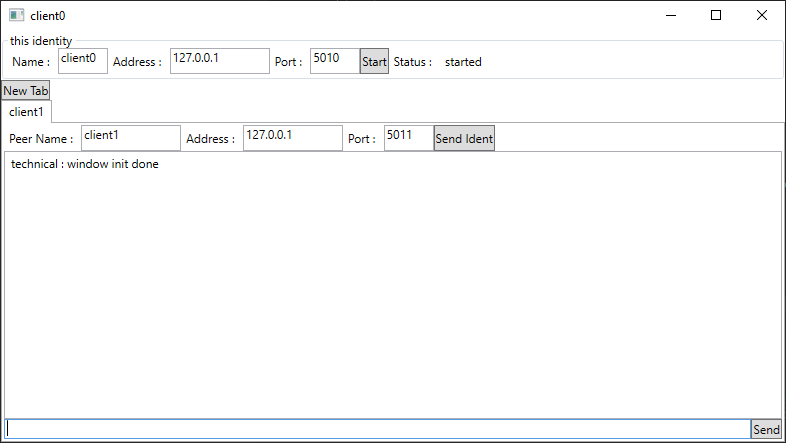
## Second case: we know the connection data of the peer

After clicking on “New Tab” button, we have now a tab named “placeholder name” …



… which once opened, allows you to enter the name, address and port of the peer you want to contact …

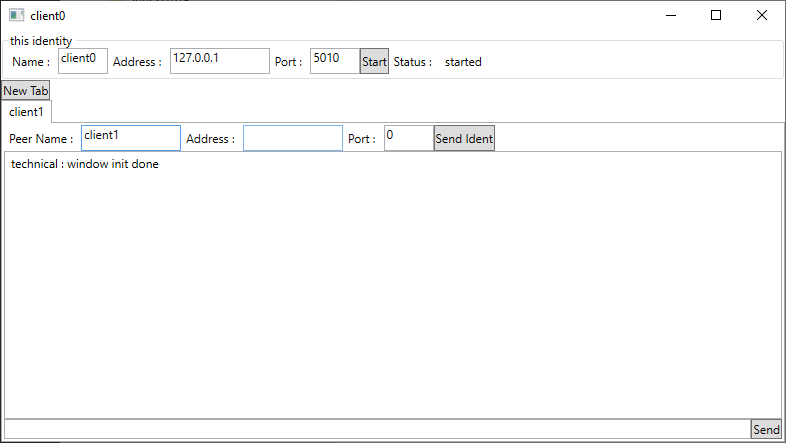




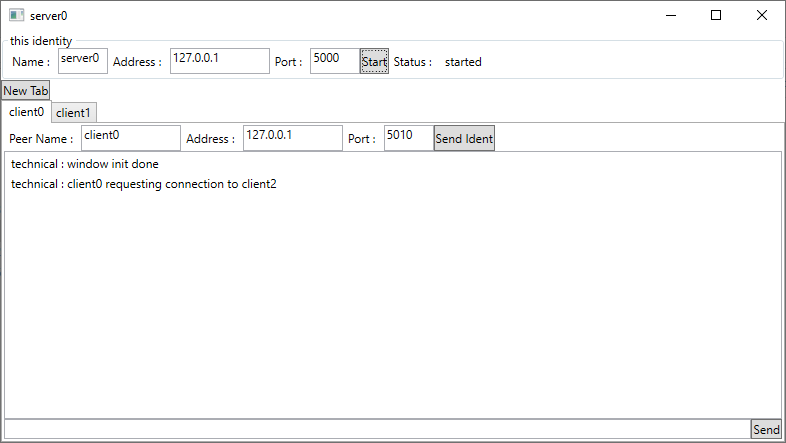
We now can send message to this peer.

## Third case: we know only the name of the peer

Same as the previous case, except that we only input the name of the peer.



When trying to send a message, the instance will ask all already known peers for the missing one, the request will look like the following screenshot:



We can see several things on this screenshot:

* Our identity is [server0@127.0.0.1:5000](mailto:server0@127.0.0.1:5000);
* We have the knowledge about 2 peers, “client0” and “client1”;
* “client0” requested connection information about “client2”, but warning, as previously said, we don’t know this peer.

# “Send Ident” button

This button is used to send an “ident” message with “this identity” information to the peer.

Why? Can we ask, it seems it is not needed as it was never mentioned in this guide?

We can imagine the following scenario: 2 instances called “peer1” and “peer2” sending messages to each other, but “peer2” crashes.

We restart the crashed “peer2” instance, start the background listening thread, but there are 2 chooses afterward:

* Manually entering “peer1” information to continue the thread of messages;
* Waiting for a message from “peer1”.

But the second choice is impossible, even if “peer1” sends a message to “peer2”, connection information for “peer1” is not known by “peer2”.

This is the case where the “Send Ident” is useful, without the need to manually enter “peer1” information in “peer2” window, clicking on this button will send all connection information to “peer2”.

The button is a helper used to remove the need of restarting all instances on the program in case of crashes. A helpful bypass for an exercise I think.

# Known limitations

* If you enter unreachable network address, it might crash;
* No automatic background thread start, because I decide to not detect “this identity” address;
* Surely many more corner cases…

# Own thoughts

* Using UDP instead of TCP is purely to simplify the socket initialization, having a connected protocol is simpler on aspect like sending a reply, integrity, etc.;
* Having an UI is lot of work, but I don’t really want to write interactive command line tools anymore, too messy and quite unreadable if multiple tasks can write on the output;
* C# helps a lot on not having to care about lifecycle of objects, synchronizing object lifecycle and socket in C++ is a lot of work, it’s better to use library like Boost, even with the drawbacks of having to use “shared\_from\_this” for example;
* I introduce a “message ack” message in the protocol useless as it is (it indicates if a message has reached its destination and if the peer send back an ack), but it helped me debugging, by crashing the app which has led to the “Send Ident” button;
* I didn’t write test for this app, since I didn’t make the “MainModel” class independent (it contains a global member of itself used by the “Peer” class), not the fact that it is difficult to do so, I just didn’t do it, so it’s impossible to have a proper unit test.