

# A Quick Start for **ffnet**

Athrun Arthur

January 21, 2013

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
1.1	Why <b>ffnet</b> . . . . .	2
1.2	<b>ffnet</b> is ... . . . .	2
1.3	Build <b>ffnet</b> . . . . .	3
<b>2</b>	<b>Use <b>ffnet</b></b>	<b>3</b>
2.1	Server . . . . .	4
2.2	Client . . . . .	5
2.3	Complete Code of PingPong . . . . .	5
<b>3</b>	<b>Inside <b>ffnet</b></b>	<b>6</b>
<b>4</b>	<b>Furthur Help</b>	<b>6</b>

# 1 Introduction

## 1.1 Why ffnet

Network programming is a very complicated thing. Of course it's simple to write a simple ping-pong network application using socket. But you have to consider many other possible situations in productive applications.

Consider ping-pong as an example. There is a server which replies pong message when receives ping message, and a client which replies ping message when receives pong message. To make the infinite loop start, the client need to send the first ping message when the connection is established. Now let's see what you need to consider if ping-pong is a product which means availability, scalability, strong and easy to extend.

- Connection management. It's obvious as there may be multiple clients. When a client is offline, the server need to know that.
- Availability. Network is complex partly because you may receive any possible messages, legal or illegal. You must distinguish those illegal messages from raw messages. This means you may need to handle some hostile attacks, like DDoS.
- Performance. It's a good practice to consider response time in network programming although ping-pong is simple. Maybe you know proactor pattern, reactor pattern, asynchronous I/O (like boost.asio), parallel programming. But you may turn a blind to these solutions because of complexity
- Maintainable. Code refactoring is a normal thing in network programming. There are many situations recall code refactoring, for example, new business requirements, unexpected network behaviors, performance tunes and security ensurance. Again, you turn a blind to possible design patterns because of complexity.
- Configurable. Another burdern to adjust very network enviornments.

There are many network libraries which aim to bring simple and powerful network programming, like boost.asio, protocol buffer from Google, ACE and mudo. But ffnet is aim to provide higher level network programming enviornment with parallel, asynchronization, security , debugging and configurable features.

## 1.2 ffnet is ...

ffnet is a opensource framework for network programming in C++. It's based on boost.asio and provide network management, package serialization and deserialization, asynchronization, security, debugging and configurable features. Now ffnet is still under heavy development.

### 1.3 Build ffnet

ffnet uses CMake to organize its source code and it depends on Boost (1.49 or higher). Suppose you have got the source code of ffnet and the directory is `/ffnet/root/dir/`. Here are the steps you need to build it.

1. `cd /ffnet/root/dir/build`
2. `cmake ../`
3. `make`

We don't recommend you to use the Boost in your system path, so you need to specify path of your Boost in

`/ffnet/root/dir/CMakeLists.txt`

. Now you can find two generated file in `/ffnet/root/dir/lib` now if you have built ffnet successfully. One is a static library and the other is a shared library.

ffnet also has some examples to show how to use it. Building examples is similar as building ffnet, that means you need to specify the necessary Boost path in the examples's CMakeLists.txt.

## 2 Use ffnet

In this section, we will write an example to illustrate how to use ffnet. The example is about ping-pong between a server and a client and it's based on TCP connection. The server is listening until a client launch a TCP connection. Once the connection is built, the client sends a *ping* message. The server will send a *pong* message as an response to each *ping* message. Then the client will get the *pong* message and re-send another another *ping* message. The progress will continue and never stop.

The first step to use ffnet is to include necessary header files.

```
#include <network.h>
```

The next step should be declaring necessary messages, ping and pong. You can declare message by inheriting `ffnet::Package` and implementing `virtual void archive(ffnet::Archive & ar);`. It's not complicated, but boring. So I prefer google's ProtoBuf and you can get more details about it from google. Now the messages are declared like this:

```
package PingPong;

message Ping{
  required string msg = 1;
  required int32 id = 2;
}
```

```

message Pong{
required string msg = 1;
required int32 id = 2;
}

```

Now it's time to initialize our network environment. In `ffnet`, network is seen as a group of nervures. Each nervure can have a TCP server, several TCP connections, and several UDP connections. Each nervure is a single thread. So the only reason to use multiple nervures is for concurrency. To use ProtoBuf in `ffnet`, we need `ffnet::ProtoBufNervure`.

## 2.1 Server

In the server side, the network is initialized like this:

```

ffnet::ProtoBufNervure pbn;
pbn.initTCPServer(10005);

```

Now we need to receive *ping* message and do something. There is no need to handle buffer and deserialization. The message `ffnet` gives us will be a deserialized *ping*. We just specify a callback handler for it.

```

pbn.addNeedToRecvPkg<PingPong::Ping>(onRecvPing);

```

Remind that `PingPong::Ping` is defined by ProtoBuf. `addNeedToRecvPkg` has the following prototype:

```

template<class MsgTy_>
void addNeedToRecvPkg(
    boost::function<void (
        boost::shared_ptr<MsgTy_>, //pMsg
        ffnet::EndpointPtr_t) > //pEP
    );

```

, in which `pMsg` is the received message and `pEP` is the source endpoint. Now we can implement `onRecvPing` and response a *pong* message.

```

void onRecvPing(boost::shared_ptr<PingPong::Ping> pPing,
                ffnet::EndpointPtr_t pEP)
{
    std::cout<<pPing->msg()<<std::endl;

    boost::shared_ptr<PingPong::Pong> pkg(new PingPong::Pong());
    pkg->set_msg("Pong from server!");
    pkg->set_id(0);
    ffnet::NetNervure::send(pkg, pEP);
}

```

Finally the left thing is to run the nervure thread.

```

pbn.run();

```

## 2.2 Client

In the client side, the network is initialized like this by specifying remote endpoint.

```
ffnet::ProtoBufNervure pbn;
ffnet::EndpointPtr_t pEP(
    new ffnet::Endpoint
        (ffnet::tcp_v4,
         boost::asio::ip::address_v4::from_string("127.0.0.1"),
         10005)
);
pbn.addTCPClient(ep);
```

Similar as server side, we need to specify PingPong::Pong's callback handler.

```
pbn.addNeedToRecvPkg<PingPong::Pong>(onRecvPong);
```

```
void onRecvPong(
    boost::shared_ptr<PingPong::Pong>pPong,
    ffnet::EndpointPtr_t pEP)
{
    std::cout<<"got pong! "<<pPong->msg();
    sendPingMsg(pEP);
}
```

Besides, we also need to send message until connection is built. ffnet provides event to do that. There are three basic events, `tcp_get_connection`, `tcp_lost_connection` and `udp_send_recv_exception`. Here we need to response `tcp_get_connection`.

```
using namespace ffnet::event;
...
Event<tcp_get_connection>::listen(
    &pbn, onConnSucc);
...
void onConnSucc(ffnet::TCPConnectionBase *pConn)
{
    std::cout<<"connect success"<<std::endl;
    ffnet::EndpointPtr_t tpp(
        new ffnet::Endpoint(pConn->getSocket().remote_endpoint()));
    sendPingMsg(tpp);
}
```

## 2.3 Complete Code of PingPong

Please check the example `pb_tcp_pingpong` in `ffnet`.

### **3 Inside ffnet**

### **4 Furthur Help**

If you have any problems, please send e-mail to [athrunarthur@gmail.com](mailto:athrunarthur@gmail.com).