ascs development documentation

ascs (<https://github.com/youngwolf-project/ascs.git>) is an asynchronous c/s framework based on asio (<http://think-async.com/Asio/>), the main purpose is to help developer to establish a c/s system quickly, it

Support tcp and udp, ipv4, ipv6 and ssl;

Support timer, object management (object pool, object reuse, object maintenance);

Maintains expandable message buffers;

Split out all logically related user protocols to packers and unpackers, developers need to provide they own packer and unpacker to support the own protocol.

# 1: A simplest example

#include <iostream>

#include <ascs/ext/tcp.h>

using namespace ascs;

using namespace ascs::tcp;

using namespace ascs::ext::tcp;

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

{

service\_pump sp;

server s(sp);

single\_client c(sp);

sp.start\_service();

while(sp.is\_running())

{

std::string str;

std::cin >> str;

if (!str.empty())

{

c.send\_msg("client says: " + str, false);

s.broadcast\_msg("server says: " + str, false);

}

}

return 0;

}

Compile it as:

g++ -o main -std=c++11 -pthread main.cpp

Then run main application, type anything with an ENTER, the client will send the message to the server (with prefix ‘client says: ‘), and the server will send the message to the client (with prefix ‘server says: ‘), PS. we only established one link between the client and the server.

So, what have been done in the ascs framework?

1. Set listening address for the server, because we didn’t set it explicitly (call ascs::tcp::server\_base::set\_server\_addr or define macro ASCS\_SERVER\_PORT) in the example, so ascs used the default value (IP is 0.0.0.0, and cannot be specified by macro):

#ifndef ASCS\_SERVER\_PORT

#define ASCS\_SERVER\_PORT 5050

#endif

1. Set server address for the client, because we didn’t set it explicitly (call ascs::tcp::client\_socket\_base::set\_server\_addr or define macro ASCS\_SERVER\_IP and ASCS\_SERVER\_PORT) in the example, so ascs used the default value:

#ifndef ASCS\_SERVER\_IP

#define ASCS\_SERVER\_IP "127.0.0.1"

#endif

#ifndef ASCS\_SERVER\_PORT

#define ASCS\_SERVER\_PORT 5050

#endif

1. ascs::tcp::server\_base bind on the specified address and begin to accept connections.
2. ascs::tcp::client\_socket\_base begin to connect the server.
3. Send messages via ascs::socket::send\_msg and ascs::tcp::server\_base::broadcast\_msg (already embodied in the example).
4. After successfully got a message, handle it, because we didn’t handle it explicitly, ascs::socket will handle it (simply print the content and the length of the message ).

PS. single\_client is a ascs::tcp::single\_client\_base, the inheritance relationships are: ascs::single\_client\_base->ascs::tcp::client\_socket\_base->ascs::tcp::socket\_base->ascs::socket;

server is a ascs::tcp::server\_base and a container, support any number of links, it use ascs::tcp::server\_socket\_base to represent links, the inheritance relationships are: ascs::tcp::server\_socket\_base->ascs::tcp::socket\_base->ascs::socket.

# 2: How to handle messages (client)?

Base on the first example (only list changed statements):

class my\_client : public single\_client

{

public:

…; //constructor, omitted

protected:

virtual bool on\_msg\_handle(out\_msg\_type& msg) {…;} //your codes

};

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

service\_pump sp;

server s(sp);

my\_client c(sp);

…; //omitted

}

# 3: How to handle messages (server)?

Base on the first example (only list changed statements):

class my\_server\_socket : public server\_socket

{

public:

…; // constructor, omitted

protected:

virtual bool on\_msg\_handle(out\_msg\_type& msg) {…;} //your codes

};

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

service\_pump sp;

server\_base<my\_server\_socket> s(sp);

single\_client c(sp);

…; //omitted

}

# 4: How to support multiple links (client)?

Base on the first example (only list changed statements):

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

service\_pump sp;

server s(sp);

client c(sp);

c.add\_socket(); //link #1

c.add\_socket(); //link #2

…; //link #N

if (!str.empty())

{

c.broadcast\_msg("client says: " + str, false);

s.broadcast\_msg("server says: " + str, false);

}

}

}

PS. client is a ascs::tcp::client\_base and a container, support any number of links, it use ascs::tcp::client\_socket\_base to represent links, the inheritance relationships are: ascs::tcp::client\_socket\_base->ascs::tcp::socket\_base->ascs::socket.

# 5: How to support multiple links (client)?

ascs::tcp::server\_base was born to support multiple links, no matter you need multiple links or not.

# 6: How to handle messages and support multiple links (client)?

class my\_client\_socket : public client\_socket

{

public:

…; //constructor, omitted

protected:

virtual bool on\_msg\_handle(out\_msg\_type& msg) {…;} //your codes

};

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

service\_pump sp;

server s(sp);

client\_base<my\_client\_socket> c(sp);

c.add\_client(); //link #1

c.add\_client(); //link #2

…; //link #N

if (!str.empty())

{

c.broadcast\_msg("client says: " + str, false);

s.broadcast\_msg("server says: " + str, false);

}

}

}

# 7: How to introduce my own packer and unpacker (two ways)?

1. Define macro ASCS\_DEFAULT\_PACKER and ASCS\_DEFAULT\_UNPACKER;

single\_client c(sp); //single link client

client c(sp); //multiple link client

server s(sp); //server

1. typedef client\_socket\_base<my\_packer, my\_unpacker> my\_client\_socket;

single\_client\_base<my\_client\_socket> c(sp); // single link client

client\_base<my\_client\_socket> c(sp); // multiple link client

typedef server\_socket\_base<my\_packer, my\_unpacker> my\_server\_socket;

server\_base<my\_server\_socket> s(sp); //server

# 8: How to call server in server\_socket (two ways)?

1. Make server to be global;
2. class my\_i\_server : public i\_server

{

public:

virtual void server\_fun() = 0;

};

class my\_server\_socket : public server\_socket\_base<ASCS\_DEFAULT\_PACKER, ASCS\_DEFAULT\_UNPACKER, my\_i\_server> {}

in my\_server\_socket, when you need to call server’s server\_fun function, just: get\_server().server\_fun();

class my\_server : public server\_base< my\_server\_socket, object\_pool< my\_server\_socket >, my\_i \_server> {}

implement pure virtual function server\_fun() in my\_server.

# 9: How to search specific link (two ways)?

1. Call object\_type find(uint\_fast64\_t id), the complexity is O(1).
2. Call do\_something\_to\_one, the complexity is O(N):

server s(sp); //also for client c(sp)

server::object\_ctype socket\_ptr;

s.do\_something\_to\_one([&](const auto& item)->bool {

//suppose get\_username() is what you’re looking for, you must implement it.

if (item->get\_username() == “abc”)

{

socket\_ptr = item;

return true; //end the loop

}

return false;

});

PS. only containers are able to find specific links, they’re server\_base, client\_base, ssl:server\_base and ssl::client\_base.

# 10: timers

1. Because almost all classes inherit from ascs::socket or ascs::object\_pool (except buffers, packers, unpackers and utilities), so they were born to be a timer, call set\_timer directly then we started a timer, please note that the timer ID must begin from super class’ TIMER\_END (the direct super class may not have this member, but ascs::socket or ascs::object\_pool does, no matter where it is, we can use direct super class::TIMER\_END to represent it, the compiler will find it for you), and if your class is inheritable, you should also define a TIMER\_END for it too, this is the policy. PS. the range of timer ID must be [0, 65536).
2. Declare a timer object, then you can use any timer IDs (but the range is still [0, 65536) ), here’s some examples (return false will end the timer, otherwise the timer will reset and start again):

service\_pump sp; //can share the same service\_pump with server and client

timer<tracked\_executor> t(sp);

t.set\_timer(0, 100, [](typename timer<tracked\_executor>::tid id) {return true;});

timer<executor> tt(sp);

tt.set\_timer(0, 100, [](timer<typename executor>::tid id) {return true;});

Here we created two types of timers, t is traceable, while tt is not, traceable timer means we can know whether all timers have stopped or been canceled (and the callback has been invoked), it’s useful when doing graceful shutdown.

1. For solution #1, the timer is traceable or not depends on whether macro ASCS\_DELAY\_CLOSE been defined to 0 or not. For solution #2, before shutdown, we need to stop all timers in t and tt.

# 11: Cygwin and Mingw support

On 32bit Cygwin, run make directly.

On 64bit Cygwin, because of asio, ascs cannot be compiled.

On Mingw64 (must with posix thread library), run mingw-build.bat directly.

On Mingw, ascs cannot be compiled (I feel it’s because it’s too old and not have been maintained for a long time, use Mingw64 instead, it supports cross compiling).