```cpp
#define TIMEOUT_SECONDS 20
#include <boost/algorithm/string.hpp>
#include <boost/beast/core.hpp>
#include <boost/beast/http.hpp>
#include <boost/beast/version.hpp>
#include <boost/asio/strand.hpp>
#include <boost/asio.hpp>
#include <iostream>
#include <string>

#include <boost/property_tree/json_parser.hpp>
#include <boost/property_tree/ptree.hpp>
#include <boost/foreach.hpp>

namespace beast = boost::beast;         // from <boost/beast.hpp>
namespace http = beast::http;           // from <boost/beast/http.hpp>
namespace net = boost::asio;            // from <boost/asio.hpp>
namespace cro = std::chrono;
using tcp = boost::asio::ip::tcp;       // from <boost/asio/ip/tcp.hpp>
using error_code = boost::system::error_code;
namespace pt = boost::property_tree;



void
fail(beast::error_code ec, char const* what)
{
    std::cerr << what << ": " << ec.message() << "\n";
}


class session : public std::enable_shared_from_this<session>
{
    tcp::resolver resolver_;
    beast::tcp_stream stream_;
    //beast::flat_buffer buffer_;
    net::streambuf buffer_;

    http::request<http::string_body> req_;
    http::response<http::string_body> res_;

    http::response_parser<http::string_body> par_;
    //net::steady_timer heartbeat_timer_;
    std::string incomplete_chunk_;
    bool is_stopped_;

public:
    // Objects are constructed with a strand to
    // ensure that handlers do not execute concurrently.
```

```
50      explicit
51          session(net::io_context& ioc)
52          : resolver_(net::make_strand(ioc))
53          , stream_(net::make_strand(ioc))
54      {
55      }
56
57      // std::optional<std::function<size_t(uint64_t, beast::string_view,
          boost::system::error_code&)>> on_chunk_body_trampoline;
58
59      void
60          run(
61              char const* host,
62              char const* port,
63              char const* target,
64              int version)
65      {
66          //{
67          //
68          //    this->on_chunk_body_trampoline.emplace(
69          //        [self = this->shared_from_this()](auto remain, auto
              body, auto ec)
70          //        {
71          //            return self->on_chunk_body(remain, body, ec);
72          //        });
73          //    this->par_.on_chunk_body(*this->on_chunk_body_trampoline);
74          //}
75          par_.body_limit((std::numeric_limits<std::uint64_t>::max)());
76
77          req_.version(version);
78          req_.method(http::verb::get);
79          req_.target(target);
80          req_.set(http::field::host, host);
81          req_.set(http::field::user_agent, BOOST_BEAST_VERSION_STRING);
82
83          // Look up the domain name
84          resolver_.async_resolve(
85              host,
86              port,
87              beast::bind_front_handler(
88                  &session::on_resolve,
89                  shared_from_this()));
90      }
91
92      void
93          on_resolve(
94              beast::error_code ec,
95              tcp::resolver::results_type results)
96      {
```

```
 97          if (ec)
 98              return fail(ec, "resolve");
 99
100          // Set a timeout on the operation
101          stream_.expires_after(std::chrono::seconds(30));
102
103          // Make the connection on the IP address we get from a lookup
104          stream_.async_connect(
105              results,
106              beast::bind_front_handler(
107                  &session::on_connect,
108                  shared_from_this()));
109      }
110
111      void
112          on_connect(beast::error_code ec,
               tcp::resolver::results_type::endpoint_type)
113      {
114          if (ec)
115              return fail(ec, "connect");
116
117          // Set a timeout on the operation
118          stream_.expires_after(std::chrono::seconds(30));
119
120          // Send the HTTP request to the remote host
121          auto self{ shared_from_this() };
122          http::async_write(stream_, req_,
123              beast::bind_front_handler(
124                  &session::on_write,
125                  shared_from_this()));
126      }
127
128      void
129          on_write(
130              beast::error_code ec,
131              std::size_t bytes_transferred)
132      {
133          boost::ignore_unused(bytes_transferred);
134
135          if (ec)
136              return fail(ec, "write");
137
138
139          do_read();
140
141      }
142
143      void do_read() {
144          if (is_stopped_)
```

```
145             return;
146
147         //auto self{ shared_from_this() };
148         //http::async_read(
149         //    stream_,
150         //    buffer_,
151         //    par_,
152         //    [this, self](error_code ec, std::size_t bytes_transferred) {
153         //        on_read_completed(ec, bytes_transferred);
154         //    });
155
156         //auto self{ shared_from_this() };
157
158         //auto self{ shared_from_this() };
159         net::async_read(
160             stream_,
161             buffer_,
162             boost::asio::transfer_at_least(4096),
163             beast::bind_front_handler(
164                 &session::read_handler,
165                 shared_from_this()));
166             //boost::bind(&read_handler, this,                              ↵
                  boost::asio::placeholders::error,                            ↵
                  boost::asio::placeholders::bytes_transferred));
167     }
168
169
170     void read_handler(const boost::system::error_code& ec, std::size_t      ↵
          bytes_read)
171     {
172         size_t offset = 0;
173         beast::flat_buffer::const_buffers_type bufs = buffer_.data();
174         //net::const_buffer bufs = buffer_.data();
175
176         std::string data(boost::asio::buffers_begin(bufs),                  ↵
              boost::asio::buffers_end(bufs));
177
178         const __int16* sp2 = boost::asio::buffer_cast<const __int16*>       ↵
              (buffer_.data());
179 //        const __int16* sp2 = boost::asio::buffer_cast<const __int16*>    ↵
        (bufs);
180         if (!ec)
181         {
182             int nSamples = 0;
183             int nBins = 0;
184             int scale = 1;
185             int bytes_used = 0;
186             while (bytes_used < bytes_read)
187             {
```

```
188                 int spectrumBytes = nBins * sizeof(__int16);
189                 if (nSamples > 0)
190                 {
191                     for (int ipp = 0; ipp < nSamples; ipp++)
192                     {
193                         if (offset + spectrumBytes > bytes_read)
194                         {
195                             std::cout << "not enough samples" << std::endl;
196                             std::cout << offset << " " << bytes_read << " " << spectrumBytes <<std::endl;
197                             nSamples = 0;
198                             break;
199 
200                         }
201                         std::cout << " * " << ipp << " ";
202 
203                         for (int i = 0; i < nBins; i++)
204                         {
205                             std::cout << (sp2[offset + i] / float(scale)) << " ";
206                         }
207                         std::cout << "\n";
208 
209                         bytes_used += spectrumBytes;
210                         offset += spectrumBytes;
211                     }
212                 }
213                 // read header and find out # of samples
214                 else
215                 {
216 
217                     std::size_t split = data.find_first_of('\x1e', offset);
218                     if (split != -1)
219                     {
220                         std::cout << " Split / offset " << split <<" / " << offset << std::endl;
221                         std::string chk = data.substr(offset, split - offset);
222                         //std::cout << "++++++++++++++++++++++++++++" << int((data.substr(split, 1)).c_str()) << std::endl;
223                         offset = split + 1;
224                         std::size_t start = data.find_first_of('\{');
225 
226 
227                         std::string chk2 = chk.substr(start, chk.size() - start);
228                         std::cout << "****" << chk2 << std::endl;
```

```
229
230                         boost::property_tree::ptree root;
231                         std::istringstream iss(chk2);
232                         boost::property_tree::read_json(iss, root);
233
234                         nSamples = stoi(root.get_child
                    ("samples").get_value<std::string>());
235                         nBins = stoi(root.get_child
                    ("sampleSize").get_value<std::string>());
236                         scale = stoi(root.get_child
                    ("scale").get_value<std::string>());
237
238                         std::cout << "# of samples: " << nSamples <<
                    std::endl;
239                         std::cout << "# of bins         : " << nBins <<
                    std::endl;
240                         std::cout << "# of bytes reads  : " << bytes_read
                    << std::endl;
241                         std::cout << "  offset           : " << offset <<
                    std::endl;
242                         std::cout << "  scale            : " << scale <<
                    std::endl;
243
244                          }
245                     }
246                 }
247
248
249
250                 std::cout << "--------------------------------" << std::endl;
251                 buffer_.consume(bytes_read);
252
253                 //do_read();
254             }
255             //else
256             //{
257             //    if (!_closing)
258             //    {
259             //        throw fatal_exception("Error on connection: " +
                  ec.message());
260             //    }
261             //}
262         }
263
264
265
266         void on_read_completed(
267             error_code ec,
268             std::size_t bytes_transferred)
```

```
269     {
270         //int done =
271         //char* tempchar = new char[bytes_transferred];
272         //boost::asio::buffer_copy(boost::asio::buffer(tempchar,
              bytes_transferred),
273         //    buffer_.data(), bytes_transferred);
274         std::cout << " ------ on read completed: bytes transferred " <<
              bytes_transferred << std::endl;
275
276         //buffer_.consume(bytes_transferred);
277
278         if (is_stopped_)
279             return;
280         boost::ignore_unused(bytes_transferred);
281
282         if (ec) {
283             do_shutdown();
284             return fail(ec, "read");
285         }
286
287         do_shutdown();
288     }
289
290     void do_shutdown() {
291         if (is_stopped_)
292             return;
293         is_stopped_ = true;
294         //heartbeat_timer_.expires_at(cro::steady_clock::now());
295
296         // Gracefully close the stream
297         //auto self{ shared_from_this() };
298         //stream_.async_shutdown(
299         //    [this, self](error_code ec) {
300         //        on_shutdown(ec);
301         //    });
302     }
303
304
305     size_t on_chunk_body(
306         std::uint64_t remain,
307         beast::string_view body,
308         error_code& ec) {
309
310         if (is_stopped_)
311             return body.length();
312
313         if (ec) {
314             do_shutdown();
315             fail(ec, "chunk");
```

```
316         }
317
318         size_t offset = 0;
319         while (offset < remain)
320         {
321             //std::cout << body << std::endl;
322             std::size_t split = body.find_first_of('\x1e', offset);
323             if (split != -1)
324             {
325                 std::cout << " Split " << split << std::endl;
326                 beast::string_view chk = body.substr(split - 2, 2);
327                 std::cout << chk<< std::endl;
328                 std::cout << "------------------------------" <<
                      std::endl;
329                 offset += split;
330             }
331             else
332                 break;
333
334         }
335         buffer_.consume(offset);
336
337         //int mPacketSamples = 0;
338
339         //while (offset < remain)
340         //{
341         //    std::cout << "offset / remain / samples:  " << offset << " /
                 " << remain << " / " << mPacketSamples <<std::endl;
342         //    if (mPacketSamples > 0)
343         //    {
344         //        // enough samples
345         //        if (offset + mPacketSamples * 2 * sizeof(__int16) <=
                    remain)
346         //        {
347         //            // do something with the IQ data
348         //            const __int16* sp = (const __int16*)(body.data() +
                      offset);
349
350         //            std::cout << "IQ " << sp[0] << ", " << sp[1] <<
                      "\n";
351
352         //            // consume all samples from the input buffer
353         //            offset += mPacketSamples * 2 * sizeof(__int16);
354         //            mPacketSamples = 0;
355         //        }
356         //        else
357         //        {
358         //            std::cout << "not enough samples" << std::endl;
359         //            break;
```

```
360         //        }
361         //
362         //    }
363         //    else
364         //    {
365         //        // search for record separator
366         //        std::size_t split = body.find_first_of('\x1e', offset);
367         //        if (split != -1)
368         //        {
369         //            std::cout << " Split " << split << std::endl;
370         //            beast::string_view chk = body.substr(offset, split - offset);
371         //            offset = split + 1;
372         //
373         //            std::cout << "****" << chk.to_string() << std::endl;
374         //            boost::property_tree::ptree root;
375         //            std::istringstream iss(chk.to_string());
376         //            boost::property_tree::read_json(iss, root);
377         //
378         //            mPacketSamples = stoi(root.get_child("samples").get_value<std::string>());
379         //
380         //            std::cout << "---" << root.get_child("samples").get_value<std::string>()  << std::endl;
381         //
382         //        }
383         //        else
384         //            break;
385         //    }
386         //}
387         //
388         //std::cout << "-------------------------------" << std::endl;
389         //buffer_.consume(offset);
390         return body.length();
391     }
392
393 };
394
395 //------------------------------------------------------------------------------
396
397 int main(int argc, char** argv)
398 {
399     // Check command line arguments.
400     if (argc != 4 && argc != 5)
401     {
402         std::cerr <<
403             "Usage: http-client-async <host> <port> <target> [<HTTP version: 1.0 or 1.1(default)>]\n" <<
```

```
404             "Example:\n" <<
405             "    http-client-async www.example.com 80 /\n" <<
406             "    http-client-async www.example.com 80 / 1.0\n";
407         return EXIT_FAILURE;
408     }
409     auto const host = argv[1];
410     auto const port = argv[2];
411     auto const target = argv[3];
412     int version = argc == 5 && !std::strcmp("1.0", argv[4]) ? 10 : 11;
413
414     // The io_context is required for all I/O
415     net::io_context ioc;
416
417     // Launch the asynchronous operation
418     std::make_shared<session>(ioc)->run(host, port, target, version);
419
420     // Run the I/O service. The call will return when
421     // the get operation is complete.
422     //
423     ioc.run();
424
425
426     return EXIT_SUCCESS;
427 }
```