There have been some serious improvements since said post and I thought a (hopefully not-too-frequent) blog-journal of the development of this particular package might be interesting/useful to some folks, especially since I’m delving into some not-oft-blogged (anywhere) topics as I use some new tricks in this particular package.

**Thank The Great Maker for C++**

I’m comfortable and not-too-shabby at wrapping C/C++ things with an R bow and I felt quite daft seeing this after I had started banging on the mosquitto C interface. Yep, that’s right: it has a C++ interface. It’s waaaaay easier (in my experience) bridging C++ libraries since Dirk/Romain’sRcpp has *so many tools* available to do that very thing.

Code

|  |
| --- |
| #ifndef MOSQUITTOPP\_H |
|  | #define MOSQUITTOPP\_H |
|  |  |
|  | #if defined(\_WIN32) && !defined(LIBMOSQUITTO\_STATIC) |
|  | # ifdef mosquittopp\_EXPORTS |
|  | # define mosqpp\_EXPORT \_\_declspec(dllexport) |
|  | # else |
|  | # define mosqpp\_EXPORT \_\_declspec(dllimport) |
|  | # endif |
|  | #else |
|  | # define mosqpp\_EXPORT |
|  | #endif |
|  |  |
|  | #include <cstdlib> |
|  | #include <mosquitto.h> |
|  | #include <time.h> |
|  |  |
|  | namespace mosqpp { |
|  |  |
|  |  |
|  | mosqpp\_EXPORT const char \* strerror(int mosq\_errno); |
|  | mosqpp\_EXPORT const char \* connack\_string(int connack\_code); |
|  | mosqpp\_EXPORT int sub\_topic\_tokenise(const char \*subtopic, char \*\*\*topics, int \*count); |
|  | mosqpp\_EXPORT int sub\_topic\_tokens\_free(char \*\*\*topics, int count); |
|  | mosqpp\_EXPORT int lib\_version(int \*major, int \*minor, int \*revision); |
|  | mosqpp\_EXPORT int lib\_init(); |
|  | mosqpp\_EXPORT int lib\_cleanup(); |
|  | mosqpp\_EXPORT int topic\_matches\_sub(const char \*sub, const char \*topic, bool \*result); |
|  | mosqpp\_EXPORT int validate\_utf8(const char \*str, int len); |
|  | mosqpp\_EXPORT int subscribe\_simple( |
|  | struct mosquitto\_message \*\*messages, |
|  | int msg\_count, |
|  | bool retained, |
|  | const char \*topic, |
|  | int qos=0, |
|  | const char \*host="localhost", |
|  | int port=1883, |
|  | const char \*client\_id=NULL, |
|  | int keepalive=60, |
|  | bool clean\_session=true, |
|  | const char \*username=NULL, |
|  | const char \*password=NULL, |
|  | const struct libmosquitto\_will \*will=NULL, |
|  | const struct libmosquitto\_tls \*tls=NULL); |
|  |  |
|  | mosqpp\_EXPORT int subscribe\_callback( |
|  | int (\*callback)(struct mosquitto \*, void \*, const struct mosquitto\_message \*), |
|  | void \*userdata, |
|  | const char \*topic, |
|  | int qos=0, |
|  | const char \*host="localhost", |
|  | int port=1883, |
|  | const char \*client\_id=NULL, |
|  | int keepalive=60, |
|  | bool clean\_session=true, |
|  | const char \*username=NULL, |
|  | const char \*password=NULL, |
|  | const struct libmosquitto\_will \*will=NULL, |
|  | const struct libmosquitto\_tls \*tls=NULL); |
|  |  |
|  | /\* |
|  | \* Class: mosquittopp |
|  | \* |
|  | \* A mosquitto client class. This is a C++ wrapper class for the mosquitto C |
|  | \* library. Please see mosquitto.h for details of the functions. |
|  | \*/ |
|  | class mosqpp\_EXPORT mosquittopp { |
|  | private: |
|  | struct mosquitto \*m\_mosq; |
|  | public: |
|  | mosquittopp(const char \*id=NULL, bool clean\_session=true); |
|  | virtual ~mosquittopp(); |
|  |  |
|  | int reinitialise(const char \*id, bool clean\_session); |
|  | int socket(); |
|  | int will\_set(const char \*topic, int payloadlen=0, const void \*payload=NULL, int qos=0, bool retain=false); |
|  | int will\_clear(); |
|  | int username\_pw\_set(const char \*username, const char \*password=NULL); |
|  | int connect(const char \*host, int port=1883, int keepalive=60); |
|  | int connect\_async(const char \*host, int port=1883, int keepalive=60); |
|  | int connect(const char \*host, int port, int keepalive, const char \*bind\_address); |
|  | int connect\_async(const char \*host, int port, int keepalive, const char \*bind\_address); |
|  | int reconnect(); |
|  | int reconnect\_async(); |
|  | int disconnect(); |
|  | int publish(int \*mid, const char \*topic, int payloadlen=0, const void \*payload=NULL, int qos=0, bool retain=false); |
|  | int subscribe(int \*mid, const char \*sub, int qos=0); |
|  | int unsubscribe(int \*mid, const char \*sub); |
|  | void reconnect\_delay\_set(unsigned int reconnect\_delay, unsigned int reconnect\_delay\_max, bool reconnect\_exponential\_backoff); |
|  | int max\_inflight\_messages\_set(unsigned int max\_inflight\_messages); |
|  | void message\_retry\_set(unsigned int message\_retry); |
|  | void user\_data\_set(void \*userdata); |
|  | int tls\_set(const char \*cafile, const char \*capath=NULL, const char \*certfile=NULL, const char \*keyfile=NULL, int (\*pw\_callback)(char \*buf, int size, int rwflag, void \*userdata)=NULL); |
|  | int tls\_opts\_set(int cert\_reqs, const char \*tls\_version=NULL, const char \*ciphers=NULL); |
|  | int tls\_insecure\_set(bool value); |
|  | int tls\_psk\_set(const char \*psk, const char \*identity, const char \*ciphers=NULL); |
|  | int opts\_set(enum mosq\_opt\_t option, void \*value); |
|  |  |
|  | int loop(int timeout=-1, int max\_packets=1); |
|  | int loop\_misc(); |
|  | int loop\_read(int max\_packets=1); |
|  | int loop\_write(int max\_packets=1); |
|  | int loop\_forever(int timeout=-1, int max\_packets=1); |
|  | int loop\_start(); |
|  | int loop\_stop(bool force=false); |
|  | bool want\_write(); |
|  | int threaded\_set(bool threaded=true); |
|  | int socks5\_set(const char \*host, int port=1080, const char \*username=NULL, const char \*password=NULL); |
|  |  |
|  | // names in the functions commented to prevent unused parameter warning |
|  | virtual void on\_connect(int /\*rc\*/) {return;} |
|  | virtual void on\_connect\_with\_flags(int /\*rc\*/, int /\*flags\*/) {return;} |
|  | virtual void on\_disconnect(int /\*rc\*/) {return;} |
|  | virtual void on\_publish(int /\*mid\*/) {return;} |
|  | virtual void on\_message(const struct mosquitto\_message \* /\*message\*/) {return;} |
|  | virtual void on\_subscribe(int /\*mid\*/, int /\*qos\_count\*/, const int \* /\*granted\_qos\*/) {return;} |
|  | virtual void on\_unsubscribe(int /\*mid\*/) {return;} |
|  | virtual void on\_log(int /\*level\*/, const char \* /\*str\*/) {return;} |
|  | virtual void on\_error() {return;} |
|  | }; |
|  |  |
|  | } |
|  | #endif |

Rcpp Modules before (that link goes to a succinct but very helpful post by James Curran) but they make exposing C++ library functionality even easier than I had experienced before. So easy, in fact, that it made it possible to whip out an alpha version of a “domain specific language” (or a pipe-able, customized API — however you want to frame these things in your head) for the package. But, I’m getting ahead of myself.

Rcpp Modules

RCPP\_MODULE(MyModule) {

  using namespace Rcpp;

  class\_<MyClass>( "MyClass")

    .default\_constructor("Default constructor") // This exposes the default constructor

    .constructor<NumericVector>("Constructor with an argument") // This exposes the other constructor

    .method("print", &MyClass::print) // This exposes the print method

    .property("Bender", &MyClass::getBender, &MyClass::setBender) // and this shows how we set up a property

  ;

  class\_<AnotherClass>("AnotherClass")

    .default\_constructor("Default constructor")

    .constructor<int>("Constructor with an argument")

    .method("print", &AnotherClass::print)

  ;

}

The mosquittopp class in the mosqpp namespace is much like the third bowl of porridge: just right. It’s neither too low-level nor too high-level and it was well thought out enough that it only required a bit of tweaking to use as an Rcpp Module.

First there are more than a few char \* parameters that needed to be std::strings. So, something like:

int username\_pw\_set(const char \*username, const char \*password);

becomes:

int username\_pw\_set(std::string username, std::string password);

in our custom wrapper class.

Since the whole point of the mqtt package is to work in R vs C[++] or any other language, the callbacks — the functions that do the work when message, publish, subscribe, etc. events are triggered — need to be in R. I wanted to have some default callbacks during the testing phase and they’re really straightforward to setup in Rcpp:

Rcpp::Environment pkg\_env = Rcpp::Environment::namespace\_env("mqtt");

Rcpp::Function ccb = pkg\_env[".mqtt\_connect\_cb"];

Rcpp::Function dcb = pkg\_env[".mqtt\_disconnect\_cb"];

Rcpp::Function pcb = pkg\_env[".mqtt\_publish\_cb"];

Rcpp::Function mcb = pkg\_env[".mqtt\_message\_cb"];

Rcpp::Function scb = pkg\_env[".mqtt\_subscribe\_cb"];

Rcpp::Function ucb = pkg\_env[".mqtt\_unsubscribe\_cb"];

Rcpp::Function lcb = pkg\_env[".mqtt\_log\_cb"];

Rcpp::Function ecb = pkg\_env[".mqtt\_error\_cb"];

The handy thing about that approach is you don’t need to export the functions (it works like the ::: does).

But the kicker is the Rcpp Module syntactic sugar:

RCPP\_MODULE(MQTT) {

using namespace Rcpp;

class\_("mqtt\_r")

.constructor("id/host/port constructor")

.constructor("id/host/port/user/pass constructor")

.constructor("id/host/post/con/mess/discon constructor")

.method("connect", &mqtt\_r::connect)

.method("disconnect", &mqtt\_r::disconnect)

.method("reconnect", &mqtt\_r::reconnect)

.method("username\_pw\_set", &mqtt\_r::username\_pw\_set)

.method("loop\_start", &mqtt\_r::loop\_start)

.method("loop\_stop", &mqtt\_r::loop\_stop)

.method("loop", &mqtt\_r::loop)

.method("publish\_raw", &mqtt\_r::publish\_raw)

.method("publish\_chr", &mqtt\_r::publish\_chr)

.method("subscribe", &mqtt\_r::subscribe)

.method("unsubscribe", &mqtt\_r::unsubscribe)

.method("set\_connection\_cb", &mqtt\_r::set\_connection\_cb)

.method("set\_discconn\_cb", &mqtt\_r::set\_discconn\_cb)

.method("set\_publish\_cb", &mqtt\_r::set\_publish\_cb)

.method("set\_message\_cb", &mqtt\_r::set\_message\_cb)

.method("set\_subscribe\_cb", &mqtt\_r::set\_subscribe\_cb)

.method("set\_unsubscribe\_cb", &mqtt\_r::set\_unsubscribe\_cb)

.method("set\_log\_cb", &mqtt\_r::set\_log\_cb)

.method("set\_error\_cb", &mqtt\_r::set\_error\_cb)

;

}

That, combined with RcppModules: MQTT in the DESCRIPTION file and a MQTT <- Rcpp::Module("MQTT") just above where you’d put an .onLoad handler means you can do something like (internally, since it’s not exported):

mqtt\_obj <- MQTT$mqtt\_r

mqtt\_conn\_obj <- new(mqtt\_obj, "unique\_client\_id", "test.mosquitto.org", 1883L)

and have access to each of those methods right from R (e.g. mqtt\_conn\_obj$subscribe(0, "topic", 0)).

If you’re careful with your C++ class code, you’ll be able to breeze through exposing functionality.

Because of the existence of Rcpp Modules, I was able to do what follows in the next section in near record time.

**“The stump of a %>% he held tight in his teeth”**

I felt compelled to get a Christmas reference in the post and it’s relevant to this section. I like %>%, recommend the use of %>% and use %>% in my own day-to-day R coding (it’s even creeping into internal package code, though I still try not to do that). I knew I wanted to expose a certain way of approaching MQTT workflows in this mqtt package and that meant coming up with an initial — but far from complete — mini-language or pipe-able API for it. Here’s the current thinking/implementation:

* **Setup connection parameters with mqtt\_broker()**. For now, it takes some parameters, but there is a URI scheme for MQTT so I want to be able to support that as well at some point.
* **Support authentication with mqtt\_username\_pw()**. There will also be a function for dealing with certificates and other security-ish features which look similar to this.
* **Make it dead-easy to subscribe to topics and associate callbacks with mqtt\_subscribe()** (more on this below)
* **Support an “until you kill it” workflow with mqtt\_run()** that loops either forever or for a certain number of iterations
* **Support user-controlled iterations with mqtt\_begin(), mqtt\_loop() and mqtt\_end()**. An example (in a bit) should help explain this further, but this one is likely to be especially useful in a Shiny context.
* mqtt[s]://[username][:password]@host.domain[:port]

You write a function with a contractually defined set of parameters and operate on the values passed in. While we should all likely get into a better habit of using named function objects vs anonymous functions, anonymous functions are *super handy*, and short ones don’t cause code to get too gnarly. **However**, in this new DSL/API I’ve cooked up, each topic message callback has **six** parameters, so that means if you want to use an anonymous function (vs a named one) you have to do something like this in message\_subscribe():

mqtt\_subscribe("sometopic", function(id, topic, payload, qos, retain, con) {})

That’s not very succinct, elegant or handy. Since those are three attributes I absolutely about most things related to R, I had to do something about it.

Since I’m highly attached to the ~{} syntax introduced with purrr and now expanding across the Tidyverse, I decided to make a custom version of it for message\_subscribe(). As a result, the code above can be written as:

mqtt\_subscribe("sometopic", ~{})

and, you can reference id, topic, payload, etc inside those brackets without the verbose function declaration.

How is this accomplished? Via:

as\_message\_callback <- function(x, env = rlang::caller\_env()) {

rlang::coerce\_type(

x, rlang::friendly\_type("function"),

closure = { x },

formula = {

if (length(x) > 2) rlang::abort("Can't convert a two-sided formula to an mqtt message callback function")

f <- function() { x }

formals(f) <- alist(id=, topic=, payload=, qos=, retain=, con=)

body(f) <- rlang::f\_rhs(x)

f

}

)

}

It’s a shortened version of some Tidyverse code that’s more generic in nature. That as\_message\_callback() function looks to see if you’ve passed in a ~{} or a named/anonymous function. If ~{} was used, that function builds a function with the contractually obligated/expected signature, otherwise it shoves in what you gave it.

A code example is worth a thousand words (which is, in fact, the precise number of “words” up until this snippet, at least insofar as the WordPress editor counts them):

library(mqtt)

# We're going to subscribe to \*three\* BBC subtitle feeds at the same time!

#

# We'll distinguish between them by coloring the topic and text differently.

# this is a named function object that displays BBC 2's subtitle feed when it get messages

moar\_bbc <- function(id, topic, payload, qos, retain, con) {

if (topic == "bbc/subtitles/bbc\_two\_england/raw") {

cat(crayon::cyan(topic), crayon::blue(readBin(payload, "character")), "\n", sep=" ")

}

}

mqtt\_broker("makmeunique", "test.mosquitto.org", 1883L) %>% # connection info

mqtt\_silence(c("all")) %>% # silence all the development screen messages

# subscribe to BBC 1's topic using a fully specified anonyous function

mqtt\_subscribe(

"bbc/subtitles/bbc\_one\_london/raw",

function(id, topic, payload, qos, retain, con) { # regular anonymous function

if (topic == "bbc/subtitles/bbc\_one\_london/raw")

cat(crayon::yellow(topic), crayon::green(readBin(payload, "character")), "\n", sep=" ")

}) %>%

# as you can see we can pipe-chain as many subscriptions as we like. the package

# handles the details of calling each of them. This makes it possible to have

# very focused handlers vs lots of "if/then/case\_when" impossible-to-read functions.

# Ahh. A tidy, elegant, succinct ~{} function instead

mqtt\_subscribe("bbc/subtitles/bbc\_news24/raw", ~{ # tilde shortcut function (passing in named, pre-known params)

if (topic == "bbc/subtitles/bbc\_news24/raw")

cat(crayon::yellow(topic), crayon::red(readBin(payload, "character")), "\n", sep=" ")

}) %>%

# And, a boring, but -- in the long run, better (IMO) -- named function object

mqtt\_subscribe("bbc/subtitles/bbc\_two\_england/raw", moar\_bbc) %>% # named function

mqtt\_run() -> res # this runs until you Ctrl-C



There’s in-code commentary, so I’ll refrain from blathering about it more here except for noting there are a *staggering amount* of depressing stories on BBC News and an equally staggering amount of un-hrbrmstr-like language use in BBC One and BBC Two shows. Apologies if any of the GH README.md snippets or animated screenshots ever cause offense, as it’s quite unintentional.

**But you said something about begin/end/loop before?**

Quite right! For that we’ll use a different example.

I came across a topic — “sfxrider/+/locations” — on broker.mqttdashboard.com. It *looks* like live data from folks who do transportation work for (which is a crowd-sourced transportation/logistics provider in India). It publishes the following in the payload:

| device:6170774037 | latitude:28.518363 | longitude:77.095753 | timestamp:1513539899000 |

| device:6170774037 | latitude:28.518075 | longitude:77.09555 | timestamp:1513539909000 |

| device:6170774037 | latitude:28.518015 | longitude:77.095488 | timestamp:1513539918000 |

| device:8690150597 | latitude:28.550963 | longitude:77.13432 | timestamp:1513539921000 |

| device:6170774037 | latitude:28.518018 | longitude:77.095492 | timestamp:1513539928000 |

| device:6170774037 | latitude:28.518022 | longitude:77.095495 | timestamp:1513539938000 |

| device:6170774037 | latitude:28.518025 | longitude:77.095505 | timestamp:1513539947000 |

| device:6170774037 | latitude:28.518048 | longitude:77.095527 | timestamp:1513539957000 |

| device:6170774037 | latitude:28.518075 | longitude:77.095573 | timestamp:1513539967000 |

| device:8690150597 | latitude:28.550963 | longitude:77.13432 | timestamp:1513539975000 |

| device:6170774037 | latitude:28.518205 | longitude:77.095603 | timestamp:1513539977000 |

| device:6170774037 | latitude:28.5182 | longitude:77.095587 | timestamp:1513539986000 |

| device:6170774037 | latitude:28.518202 | longitude:77.095578 | timestamp:1513539996000 |

| device:6170774037 | latitude:28.5182 | longitude:77.095578 | timestamp:1513540006000 |

| device:6170774037 | latitude:28.518203 | longitude:77.095577 | timestamp:1513540015000 |

| device:6170774037 | latitude:28.518208 | longitude:77.095577 | timestamp:1513540025000 |

Let’s turn that into proper, usable, JSON (we’ll just cat() it out for this post):

library(mqtt)

library(purrr)

library(stringi)

# turn the pipe-separated, colon-delimeted lines into a proper list

.decode\_payload <- function(.x) {

.x <- readBin(.x, "character")

.x <- stri\_match\_all\_regex(.x, "([[:alpha:]]+):([[:digit:]\\.]+)")[[1]][,2:3]

.x <- as.list(setNames(as.numeric(.x[,2]), .x[,1]))

.x$timestamp <- as.POSIXct(.x$timestamp/1000, origin="1970-01-01 00:00:00")

.x

}

# do it safely as the payload in MQTT can be anything

decode\_payload <- purrr::safely(.decode\_payload)

# change the client id

mqtt\_broker("makemeuique", "broker.mqttdashboard.com", 1883L) %>%

mqtt\_silence(c("all")) %>%

mqtt\_subscribe("sfxrider/+/locations", ~{

x <- decode\_payload(payload)$result

if (!is.null(x)) {

cat(crayon::yellow(jsonlite::toJSON(x, auto\_unbox=TRUE), "\n", sep=""))

}

}) %>%

mqtt\_run(times = 10000) -> out

[](https://i1.wp.com/rud.is/b/wp-content/uploads/2017/12/tojson.gif?ssl=1)

What if you wanted do that one-by-one so you could plot the data live in a Shiny map? Well, we won’t do *that* in this post, but the user-controlled loop version would look like this:

mqtt\_broker("makemeuique", "broker.mqttdashboard.com", 1883L) %>%

mqtt\_silence(c("all")) %>%

mqtt\_subscribe("sfxrider/+/locations", ~{

x <- decode\_payload(payload)$result

if (!is.null(x)) {

cat(crayon::yellow(jsonlite::toJSON(x, auto\_unbox=TRUE), "\n", sep=""))

}

}) %>%

mqtt\_begin() -> tracker # \_begin!! not \_run!!

# call this individually and have the callback update a

# larger scoped variable or Redis or a database. You

# can also just loop like this `for` setup.

for (i in 1:25) mqtt\_loop(tracker, timeout = 1000)

mqtt\_end(tracker) # this cleans up stuff!