## Setting Up the Build Infrastructure

The first step was to decide on a build system. GNUMakefiles were an option, however, they are cumbersome and not particularly well suited multi-level source trees (or at least is considered bad practice)[<http://aegis.sourceforge.net/auug97.pdf>]. There’s also the matter of being able to actually write them, and write them well. So traditional GNUMakefiles were removed from consideration.

Next was using the autotools collection. Instead of having to write complicated makefiles, a large portion of it can be abstracted out into smaller autotools compliant lists with a toplevel auto tools “script” that can be used to generate all the makefiles needed in each wanted directory. However, autotools is a very, very old build system and appears more complicated (and cumbersome) than make itself. Experience is also very limited regarding its use and would take far too long to make it reliable and stable to be used for the project.

Finally, a program called cmake was settled upon. It offers the abstraction of autotools but made much, much simpler. Each directory contains a CmakeLists.txt file that tracks special commands, subdirectories, targets, unittests, depencies, etc. Many newer libraries have started using Cmake - even though its thirteen years old, it has only just begun to gain traction. It offers human readable list files, a variety of “language” features and, if they desire, the user can even change the build engine (i.e. Ninja, GNUMakefiles, MinGW Makefiles, etc). It’s also directly compatible with modern IDEs such as Clion from JetBrains.

## Setting Up Gtest/Gmock

As soon as the project was started, it was decided that unit tests were a necessity. It would help in tracking down problems now, and it meant that small errors were corrected as soon as the library was built, not when the entire system fails and there’s no way to find out where.

After experimenting with building cppunit and Boost.Test, it was realised that mocking would also be a very useful tool to have, and neither of those projects supported it. Eventually GoogleTest and GoogleMock were settled upon.

This made testing a great deal easier as it meant that there was a large user base and a strong customer base behind it. It also meant the unit test and mocking capabilities were built “into” each other, and making sure the test and mocking frameworks worked together comfortably (and uniformly across files).

The GitHub project also provided some CMake targets that automatically downloaded, built, and linked them with my unit tests. Meaning that the dependency was self-resolving (requiring the more prudent users, who wish to compile the unit tests, need only an internet connection).

## Project structure

Originally, the project structure was very flat. The top-level directory contained a single CMakeLists.txt and the main.cpp (for ProtDev itself). With each library/component of the program in a folder in there. Each folder then contained its own tests/ folder that held the unit test files. This method became cumbersome and very hard to maintain. So, in response more research went into the best way of handling a CMake project. Eventually the structure currently in use was found as per:[<https://rix0r.nl/blog/2015/08/13/cmake-guide/>]. Instead of every single package needing to be referenced in the top-level CMakeLists.txt Every single library is its own self-contained project that is exported to the larger CMake Project. It also allows for more implementation hiding between libraries. And it helps to keep all executable files much tidier (along with executable targets being a great deal simpler to define).

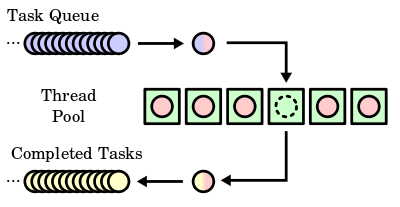
## Moving From Raw Pointers To Special Pointers

A new feature of C++11 and up is the addition of *special pointers*. These special pointers are reference counted (thus are deleted automatically when nothing is referencing them) and allow finer control over how they should behave as “pointers”. When it was realised that modern C++ advocates *against* the use of traditional raw pointers, it was set about updating what had been written to conform to the current standard.

This was a very difficult task as at the time a great deal of code had already been written and special pointers were still a new concept. However, having fully updated every part of the code to conform, the project has been left in a much more conforming and robust state. Memory leaks will be much rarer however the code is a bit more complicated.

## Generic Thread Pool

The most significant feature of ProtDev is its ability to handle multiple threads sending data to a target. For increased control and higher bandwidth generally. The original plan was to have threads handled on an as-required process. Whatever object required them would just be handed one. After some more deliberation, discussion and research, the Thread-Pool Queue pattern was discovered and implemented.



1: Picture from Wikipedia (<https://upload.wikimedia.org/wikipedia/commons/thumb/0/0c/Thread_pool.svg/600px-Thread_pool.svg.png>)

This method allowed the maximum number of threads to be limited from a central area. However, a thread pool for every possible return type a thread may have. One method would be to just do that, and create a new handler for each return type, another would be to do away with type safety and use a void\* return type.

C++11 has seen a significant change in C++ and how it is to be used and implemented. This includes the “auto” specifier and other functions dedicated to more generic programming that tie in with C++’s <*template*> functionality and to push more work onto the compiler to catch more potential run-time issues.

Auto only works if the compiler can figure out what the type will be upon the instantiation of the variable; however, using the *decltype* specifier from C++11 along with C++14’s ability to deduce a return type depending on template parameters, a totally generic method can be written that decides the types on compile-time, thus saving lines of code and maintenance effort, ultimately less code smells.

With all the tools available, in terms of the modern C++ standard, a single thread pool was created. Cutting maintenance drastically and creating a good robust solution that can be used across the system without too much specificity. This will also help with system configuration as it means the maximum threads can be defined more specifically.

## Network Communication

The C++ standard library, though incredibly powerful and featureful, still leaves areas to be desired. Though C++17 is supposed to be addressing these, there’s no proper documentation on it yet (nor is it fully implemented in the compiler in use, GCC). The basic communication handler will have to be written using the old BSD Sockets library (as they were in C on Linux). This also means that functions (as opposed to methods) need to be used, which can cause issues with the googletest/googlemock unit testing framework.

It was decided that the absolute first step in this would be to create a class wrapper around the BSD sockets library and pass that in as a parameter to the class. This way a class for mocking outputs/inputs can be written that can do things in a more OO way.

To verify results, a simple hellogoodbyeworld program is being written. The idea behind it is that it can be run and then ProtDev against it. It’ll run a very simple protocol that simply answers back when spoken to.

## Logger and Testing

The first thing that was implemented was the logger service. This was to better understand some aspects of the problem and to delve directly into C++’s new threading features. The logger has gone through many iterations. The original plan for the logger consisted of having the logs written to using a single thread that sat in a full “log handler”. Each log file object would be passed into whatever component needed it and it would then write to that. The log handler would then loop over each log file and flush its messages to the stream chosen for that log file.

The original logger library was deleted in favour of re-writing it with fresh eyes and more experience into its current state. A great deal of chaff was skimmed out and more significance was placed on using C++ features such as std::condition\_variable; a kind of mutex that allows users to notify other threads with the same variable that they may continue, sleeping them in between calls and waits.

It was also decided that a logger could instead be setup to use more generic streams and repurposed for outputting to *stdin* or *stderr* as well as the file streams they were intended for. The best way fo implemtning this was using a strategy pattern that allows the component to *choose how* it wants its output to be displayed*.*

To try saving on global singletons some research was done into the best way of implementing this. Eventually it was discovered how best to use a singleton [<https://stackoverflow.com/questions/5877779/c-logger-class-without-globals-or-singletons-or-passing-it-to-every-method>] that allows us to print to the log whenever, wherever, without having to pass loggers around the entire system.

## Time Keeping/Timer

It was clear from the start that some way of tracking the time would be very important, however due to a need to get a working prototype the prioirty of the timer was demoted considerably, leaving it as a side task.

The timeTicker class has gone through many iterations using C++’s own std::chrono library and using timeval/timespec. Eventually, a solution using both timeval *and* timespec was created (each for different precisions). Std::chrono was not used as it works better when the time does not need to be outputted into any sort of string format.

## Plugin Architecture

Plugins are a major feature of the project. As such it is very important that the loader is robust, well tested and easy to use.

When performing the original legwork, in the form of isolated “spikes” (self-contained programs that were used for testing) that were used to verify that different parts of the software were at all possible or simply to see how new libraries work, tests were done using the *dlfcn.h* header. And originally the tests were successful, however when applied to a more complex project that required passing C++ objects between the boundary things quickly decayed and it was found that casting between void\* and function pointers in C++ is not allowed (if it were to be done correctly to the standard).

So, instead the boost.dll library was used as a robust and portable way of dynamically loading shared objects into the program. The caveat with this rested with boost not being totally compliant with C++1x, especially relating to shared and unique pointers, having to use the boost variants as opposed to the C++ ones.

[[https://stackoverflow.com/questions/12314967/cohabitation-of-boostshared-ptr-and-stdshared-ptr#comment28669000\_12315035](https://stackoverflow.com/questions/12314967/cohabitation-of-boostshared-ptr-and-stdshared-ptr" \l "comment28669000_12315035)]

However, using one of the contructors of the shared\_ptr we move the boost::shared\_ptr into a std::shared\_ptr and vice-versa. While this may add an extra step when switching between libraries, it will certainly help with keeping things more uniform and, more importantly, standard within the software.

Boost came with its own complications. Especially regarding C++ standards. Due to the genuine requirement to use the latest and greatest parts of the C++ standard, the packaged boost libraries needed to be rebuilt to make use of this new standrd and to actually build correctly. This was, eventually, successful by passing “cxxstd=17” to the “./b2” command upon building [<https://gist.github.com/dennycd/5890475>].

This caused many of the boost libraries already present to break and cause dependency issues on the PC the project is being develped on. It was eventually resolved only by removing the offending, newly built, packages. Re-installing boost through the package manager. Then rebuilding the special version of boost needed to build the project and then retargeting the CMake module to pick up the correct version of the libraries for building against the project.

Once a stage was reached where it made sense to perform a build, several template issues were thrown up. Issues arose around what is returend by the import alias function. The sharedMap\_t<> was changed severalt imes to reflect changes along with the import\_alias function. However, after countless configurations, it was discovered that the template of the import alias should contain only what the factory method itself returns and not other types (such as the boost/std::function in the import template). Areas that contained too many types in templates were also discovered.

Eventually a build was performed and was successful, however it required rethinking many approaches to the templating in the segment.

## Per Library README.md

Fairly late into development it was realised that each library should have a README.md file to help explain its purpose and how best to include it into an external project.

The README files were eventually decided to simply state:

* the purpose of the library
* any particular ‘gotchas’ about the library
* how to use the library, including how to include it into another library or project
* AOB concerning the library and its unit tests

This step proved very useful when referring back to older libraries and proved as a simplistic code review to make sure that each library’s interface still made sense.

## Parsing Complications

After many false starts on attempting to creating a lexer/parser It was realised that that was beyond the scope of the project and not beneficial. After some research, the header-only library rapidxml was discovered. [<https://stackoverflow.com/questions/9387610/what-xml-parser-should-i-use-in-c>] It is fast and simple and does everything required of it without too much bloat.

After time in a seperate spike understanding rapidXML and how it works, it was decided that it would be best to abstract the calls to rapidXML out to another class. While this is required for the purposes of unit testing and mocking, it also provides a platform that allows the parser to be swapped out at a later date without having to change the code in the gubbins of the project.

During development and feature verification it was also discovered that rapidxml’s implementation of std::exception (rapidxml::parse\_error) provides a *where()* method for detailing where an error has been found. It returns a pointer to the location that can then be expanded by the user upon a failed analysis to detail where the parse error occurred. Thus, requirement 5.2.1 is immediately ticked off and making the implementation for that much easier.

While developing, it was decided that the depth of the node that is selected would be controlled via a “queue”. However testing showed that that was totally wrong and it should have been a stack, a simple fredian slip as opposed to a genuinly faulty design.

## Running a test

One of the final libraires that need to be created was that of the test runner itself. It consists of two components. The TestRunner “handler” class and “TestThread” objects. The TestRunner will setup the environemnt, create the test threads and finally run them.

Test threads will actually send the data using the specified communication using the created protocol and run for a certain period or until they’re cancelled/killed. Each test thread handles its *own* life-time duration by a ratio of the (the rate per second)/(number of threads).

Currently the software only has the “scaffolding” for more of the advanced features that need to be implemented due to time constraints and the everso important need for the code to be easily extendable for future features, developers and maintainers to tinker with it.

## First run

As soon as the testRunner was completed, a minimal test was done to make sure that what was currently implemented works to some degree. A simple executable was mocked up.

Immediately the software threw. The plugin library claimed that the SO was in an incorrect ELF format. This was because there was no file extension check. Having added that check another run was performed which instead caused a segfault as soon as any method within the plugin is called.

## API

The API was one of the harder parts of the program as it was using technologies more foreign concepts than previously dealt with. It also required a lot of third-party libraries to make work; both in the API itself and in the communication using Google’s protobuf technology.

The first step was in figuring out the best way to package the required libraries or potentially to just distribute them with the software itself (to make building easier later on).

GRPC can be included as a part of the source in the “otherLibraries” directory which, with some tinkering, can handle it’s protobuf dependency too.

The next issue was using protobuf to generate classes based on the protobuf format that will be used to request information through gRPC and then integrating those source files into my own source tree.