CSC 499 (Winter 2020) – Status Report Three

Michael Windels
March 13, 2020
V00854091

Abstract

This document summarizes the parts of this project that I have been working on for the past three weeks. These include abstractions related to log entries, inter-node communication, and paxos round identification. This document also covers some of my findings and feelings about the D programming language now that I have been working with it for the last two months. Finally, the document ends with a brief statement on what I plan to do in the last three weeks of this project.

1 What is Complete

Originally, my plan was to have a working implementation of multi-paxos done by this point. However, as the project developed, it became clear that additional abstractions would be necessary to support my implementation. Accordingly, I have spent the last three weeks identifying and implementing such abstractions. The abstractions which I identified as most salient are those which help to:

1) Store multi-paxos log entries.
2) Communicate between nodes in a quorum.
3) Uniquely identify multi-paxos rounds.

To solve issue (1), I have devised a templated log entry type which can represent any of the mutable member functions of its template parameter. This includes storing a string which uniquely identifies the function (generated by the linker) and a tuple which stores all of the function's arguments except the hidden `this` argument. One crucial aspect of this type is that it must have a uniform size so that it can be stored in an array (the log). To accomplish this, the log entry type uses a union that encapsulates all possible argument tuples for the member functions of some given data type. This union type is constructed at compile-time using D's powerful compile-time reflection and code-injection capabilities. Because a union is a sum type, its memory footprint is only equal to the memory footprint of the largest argument tuple. This holds regardless of how many argument tuples it accommodates. By using a union rather than a pointer to space allocated elsewhere, we also keep our data contiguous and ensure a higher likelihood of cache hits when iterating over the log. However, using a union does prevent the log entries from representing variadic functions, as they do not have a fixed number of parameters. Furthermore, the log entries cannot store template functions, as D's compile-time reflection tools only recognize their types as `void`.

For problem (2), I have put together an RPC-like framework for communicating between nodes. Under other circumstances, I would have used an existing RPC framework rather than developing my own. In this case, however, the framework must be able to serialize arbitrary data structures into binary representations. This is necessary so that function arguments may be passed between nodes. While there exists a serialization library called Cerealed (see [1]) which can accomplish this, no DLang RPC libraries boast such powerful serialization capabilities. As a result, I have developed my own framework to handle inter-node communication. The framework uses two mechanisms (included on each node) to communicate. The first is the connection pool, a pool of open sockets connected to various addresses that are leased out upon request. This is the active (or "calling") side of the connection. The other mechanism is the terminal pool, which accepts incoming connections and maintains its own pool of previously opened sockets. This is the passive (or "responding") side of the connection. Using socket pools means that we do not have to regularly tear-down and re-open connections to the same machines every time a message is sent. Currently the terminal pool is essentially complete, while the connection pool is still in the works. I expect the latter to be complete by the end of the week. Figure 1 illustrates the relationship between the connection pool and the terminal pool.

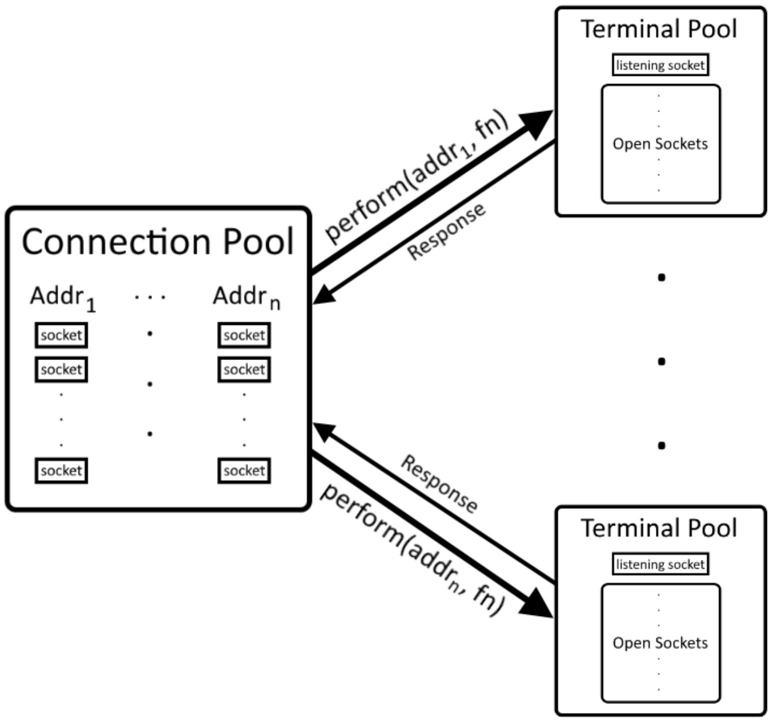

Figure 1 – The connection pool and associated terminal pools.

Of the three issues, issue (3) was the easiest to solve, but no less important than its peers. A correct implementation of multi-paxos requires that every round of prepare and accept messages must be uniquely identifiable by a proposal number. A simple integer will not solve this issue, as each node must be able to produce unique proposal numbers even when partitioned from its peers. To resolve this, I have implemented an ID type composed of two natural numbers. One is the round number, while the other uniquely identifies the node that generated it. Since every node generates ID values using their own unique identifiers, no two IDs will be equal if they were generated on separate nodes. Proof of this can be demonstrated by showing that the set of all IDs has a one-to-one mapping to the set of natural numbers. First, let there be n nodes in the cluster. Then, for any ID value $i = (round, node)$ where $round$ is a natural number, and $0 \leq node < n$, let i map to the natural number $n(round)+node$.

Because *round* can be any natural number, $n(round)$ will be any natural number that is a multiple of *n*. Between every $n(round)$ there is a gap of *n* – 1 numbers. Since *node* is in the range [0, *n*), adding *node* to $n(round)$ will always produce a value in the range $\left[n(round),\ (n+1)(round)\right)$ for all values of *n* and *round*. Thus, all ID values map to unique natural numbers, meaning that all ID values are unique. No two IDs generated on separate nodes will be equal because the differing *node* values make them map to different natural numbers. Because of this mapping, we also know that all ID are totally ordered. This means that our IDs can be used to distinguish old proposals from new proposals, a necessary feature for paxos. This notion of the proposal number as a pair of *round* and *node* numbers is borrowed from [2], though the proof of their uniqueness and orderedness is my own.

2 Working with D

Implementing the features described in section (1) has been a great opportunity to learn a lot about a language of which I previously knew little. While there are features of the language which I enjoy and appreciate, there are others which I have found insufficient and occasionally stifling. This section touches upon some of these features.

2.1 What I Like

There are plenty of things to like about D. Coming from a background of C-like languages, working in D feels familiar. At the same time, D offers many powerful new features that fundamentally change the way some problems can be tackled. The two features most relevant to this project (and most radically different from C) are compile-time reflection and compile-time code injection. Without these two features, this project may have been a non-starter. From the interface layer of the distributed type, right down its log entries, reflection and code injection have been instrumental in making the distributed type compatible with the types it is distributing.

D also offers plenty of comparatively simpler quality-of-life features that make the language easier to use. Among these are scope guards, template constraints, and garbage collection. Scope guards make the Resource Allocation Is Initialization (RAII) idiom from C++ easy to accomplish without having to create new types to ensure that resources are properly freed during stack unwinding. Template constraints allow the programmer to specify which template parameter types are valid. Compared to the analogous Substitution Failure Is Not An Error (SFINAE) idiom in C++, template constraints are both simpler to impose and simpler to read. Finally, D's garbage collection also proves helpful in most situations by automating lots of cleanup that is otherwise easily overlooked.

2.2 What I Like Less

Although there is plenty to like about D, I have also found the language lacking in a few respects. For example, concurrency in D can feel clunky at times. D's approach to message passing is akin to Go, but it only provides one channel per thread. Consequently, every time a user invokes the `receive` function in some thread, they must handle all possible types of incoming messages. This is especially troublesome in the main thread. Another issue with concurrency in D is the `shared` qualifier. This qualifier is meant to inform the compiler not to store the data in thread-local storage (which it would otherwise do by default). However, it also substantially limits what kinds of operations can be performed on the data. The `shared` qualifier can be cast away, but doing so frequently can quickly clutter up code, making it more difficult to read.

Another issue with D is standard library types that have essentially been copied verbatim from C. Such types lack properties that would otherwise be idiomatic in D. One such example is the `SocketSet` data type. Although a `SocketSet` is simply a set of sockets, it is not iterable like an array or an associative array. Instead, to iterate over a `SocketSet`, the user must compare its contents against the contents of some array containing the values inserted into it. Despite these issues, I have still found it enjoyable to work in D. I believe that the unique features offered by the language provide compelling use cases in spite of the language's shortcomings.

3 What is Upcoming

The foundation is set, now it is time to bring everything together. In the final three weeks of this project, I hope to tie all of the abstractions together into an implementation of multi-paxos. When the framework is sufficiently complete, I will use Google Compute Cloud to help test it distributed across multiple nodes. When possible, I will continue looking into testing with Jepsen.

4 Sources

[1] Cerealed Repository: https://github.com/atilaneves/cerealed

[2] Paxos Video Resource: https://www.youtube.com/watch?v=JEpsBg0AO6o