Towards a Multi-Paxos Framework in D

CSC 499 – Final Report

Michael Windels
April 3, 2020
V00854091

For CSC 499 (Winter 2020) at UVic.
Supervised by Dr. Yvonne Coady.
Repository: https://github.com/MWindels/auto-distribute

0 Introduction

The goal of this project was to implement a framework which would distribute arbitrary data structures using Multi-Paxos. Although this might imply that the project's main focus would be distributed consensus, it turned into something much broader in scope. Exploring distributed consensus algorithms on their own (without regard for what kind of data structure is being distributed) can abstract away many important implementation details. The form and function of the data structure being distributed can enable optimizations, but also impose limitations. Designing a consensus framework which accepts *any* type requires that substantial work be done upfront to develop type-agnostic interfaces, containers, and protocols before one can even address the consensus problem. The D programming language has made this initial work easier compared to other C-like languages, but by no means unchallenging. This report explores the issues associated with developing a type-independent consensus framework, along with the design choices and limitations made along the way. Though my implementation of Multi-Paxos is not yet complete, this project shows that a general-purpose consensus framework is absolutely feasible subject to some stipulations.

For reference, the implementation can be found at:

https://github.com/MWindels/auto-distribute

1 What Does it Take to Distribute Any Type?

At its most fundamental, Multi-Paxos can be interpreted as a means to maintain the consistency of a replicated log. This description handily circumvents the structure of the data being distributed. Using a log, all distributed data can be simplified down to a series of operations applied to some initial state. However, issues with the log abstraction begin to arise with its entries. A log entry must be able to represent any operation performed on the distributed data. If the structure of the data is well-defined, this might be easy. A key-value store, for example, may only have two operations: get and set. In order to distribute an arbitrary type, one must know what operations can be performed on that type. A distributed consensus framework must therefore be able to programmatically (that is, without user input) reflect on the data structure which is being distributed. Using reflection, one can ascertain the member functions of a type. Once these operations are known, then one can develop log entries which can represent a reasonable set of possible operations on the type being distributed.

The ability to reflect on the data type being distributed is also important because it allows one to represent the results of operations in a way which is the same for every operation on the type. This is important because it allows one to encapsulate all possible results of an operation as a single type with a fixed size. This is useful for several reasons. Such result packages can be stored in the log alongside their associated log entries. This helps ensure that the same operation is never performed more than once, even if the requester failed and had to re-try the operation. Additionally, single-type fixed-size results can also be sent and received across the network with ease, because their exact size is always known.

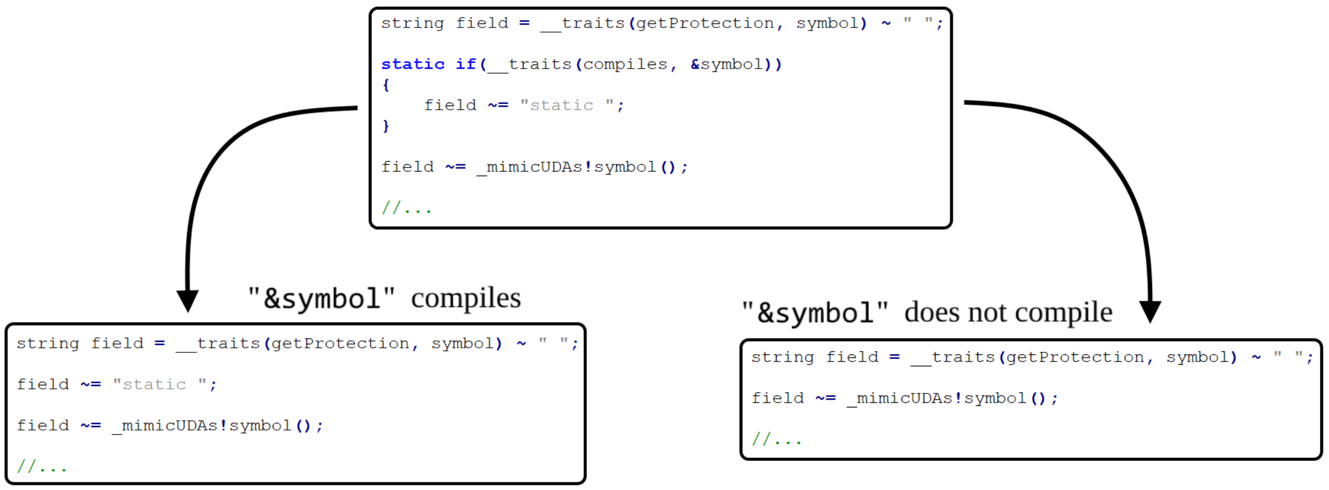
Reflection provides one more crucial capability: it allows the consensus framework to mimic the structure of the type being distributed. This includes all of the member functions, fields, types, and aliases defined in the type. This mimicry is useful to the user because it means less of their code which interfaces with their type needs to change to accommodate its distributed version. However, it is also of great importance because mimicked member functions provide an excellent place to inject consensus logic. The same could not be accomplished with inheritance, at least in D. If the distributed type inherited the structure of a non-distributed type, then consensus logic could be bypassed by marking members with the `final` keyword (which prohibits overriding). Furthermore, we would not be able to make changes to the fields of a type if they were inherited. This too could enable the user bypass consensus logic.

Clearly reflection is not only a convenient feature, but an integral asset if one is to devise a type-agnostic distributed consensus framework. The D language's support for (compile-time) reflection has been the single most useful aspect of the language for this project. It is of such importance to this project that I do not believe that the project could have been accomplished in a language like C, which lacks built-in reflection capabilities.

2 From Consensus Problem to Meta-Programming Problem

Though reflection has been an important aspect of this project, conditional compilation and code injection have been equally important co-requisites. In fact, without conditional compilation and code injection, reflection would be of little use. While reflection allows one to analyze the type being distributed, conditional compilation and code injection allow one to take the information gleaned from reflection and use it to construct the distributed type. For this reason, the log entries, result packages, and mimicked interfaces discussed in the first section all use conditional compilation and code injection extensively.

In the repository, any code segments which include the statements “`static if`” and “`static foreach`” are performing conditional compilation. In D, these are the compile-time equivalents of `if` and `foreach`, just as `static assert` is the compile-time equivalent of `assert`. Although, it is worth noting that unlike their run-time counterparts they are not control-flow constructs, but code-generation constructs. Thus `static if` and `static foreach` incur no additional cost at run-time. However, the data supplied to such constructs must be available at compile-time. In C, preprocessor directives like `#if`, `#ifdef`, and `#ifndef` serve a similar purpose. Figure 1 provides a simple example of how this project makes use of D's conditional compilation.



```
string field = __traits(getProtection, symbol) ~ " ";

static if(__traits(compiles, &symbol))
{
    field ~= "static ";
}

field ~= _mimicUDAs!symbol();

//...
```

"`&symbol`" compiles

```
string field = __traits(getProtection, symbol) ~ " ";

field ~= "static ";

field ~= _mimicUDAs!symbol();

//...
```

"`&symbol`" does not compile

```
string field = __traits(getProtection, symbol) ~ " ";

field ~= _mimicUDAs!symbol();

//...
```

Figure 1 – A code snippet where what code is compiled depends on a conditional.

Segments of code which include the "`mixin`" statement are performing code injection. While conditional compilation is helpful for generating code depending on some Boolean condition (`static if`) or the contents of some compile-time sequence (`static foreach`), code injection is far more versatile. With code injection, one can construct expressions parameterized by compile-time constants which would not otherwise be syntactically sound. One can even leverage D's Compile Time Function Evaluation (CTFE) to generate arbitrarily large code segments. Figure 2 demonstrates two code snippets with `mixin` statements. The first uses CTFE to inject a (potentially) large amount of code, while the second mixes in two comparatively simple statements that would not otherwise parse.

```
union Params
{
    mixin(forAllMembers!(T, _createParamField, nop, nop)());

    //...
}
```

```
Params arguments;

//...

mixin("arguments." ~ ConsistentMangle!symbol) = Tuple!(mixin("arguments." ~ ConsistentMangle!symbol ~ ".Types"))(args);
```

Figure 2 – One `mixin` using a CTFE and two others used to inject a compile-time constant.

The common thread that ties both conditional compilation and code injection together is that they both generate code. The `static if`, `static foreach`, and `mixin` statements are all meta-programming constructs, because they are pieces of code which generate code (hence *meta*-programming). The central challenge of this project has been using meta-programming to concretely define type-dependent parts of the framework, while allowing type-independent parts to function without concern for what type is being distributed. In combination with reflection, meta-programming constructs were used to create many parts of this project. For example, the log entries and result packages were created using meta-programming techniques. This allows the consensus logic to mostly ignored the data they contain. Additionally, mimicking the interface of the type being distributed also involved a great deal of meta-programming. When this project was starting out, I did not foresee just how integral meta-programming would be to this project. Surprisingly, developing a consensus framework has been more of an exercise in meta-programming than an exercise in distributed consensus.

3 The Project's Other Problem Domains

Of course, this project involved more than just meta-programming. This section explores two of the issues most salient to this project, other than meta-programming.

3.1 Consensus

Naturally, consensus issues played a prominent role in the project. One such issue was learning about Multi-Paxos. While Leslie Lamport provides a rigorous and approachable definition of the (single-value) Paxos algorithm in [1], his description of Multi-Paxos (in section three of [1]) is much more vague. For example, Lamport mentions leader election in [1], but does not prescribe a means to elect a leader. This project uses a Raft-like approach to leader election, complete with voting and terms. Although unlike Raft, the term number also imparts information about the candidate being voted for (for details refer to the `ID` data structure at `src/distribute/id.d`, or status report three).

However, it turns out that leader election in Multi-Paxos is only an optimization. One could implement Multi-Paxos without leader election by simply running single-value Paxos for every entry in the log. Though this is wasteful for two reasons.

First, it sends a lot of useless PREPARE messages. If some acceptor node is already in agreement with the proposing node on the state of the log, then having the acceptor PREPARE is a waste of time. Proposers issue PREPARE messages in order to learn about any previously accepted values. Electing a leader in Multi-Paxos forbids other nodes from acting as proposer. Thus, once an initial round of PREPARE messages are complete for a majority of nodes, then the leader knows the contents of its followers' logs and does not have to send PREPARE messages, only ACCEPT messages (until leadership changes).

The other reason why leaderless Multi-Paxos is suboptimal is that every node must perform a round of Paxos in order to determine which values are chosen for each log entry. In this context, “chosen” means that the value has been accepted by a majority of nodes. For any given log entry, only the proposer knows whether that value has been chosen, because only the proposer knows whether a majority of nodes accepted the ACCEPT message issued for that entry. With a leader, this issue is solved by simply redirecting all queries to the leader (who is the proposer of all log entries for their term). In the event that a leader crashes, the new leader could get back up to date by issuing PREPARE and ACCEPT messages for every entry in the log about which it is uncertain. In practice, one would use another kind of message to periodically inform non-leader nodes that values for certain log entries have been chosen. For a comprehensive exploration of the challenges associated with implementing Multi-Paxos, see [2].

3.2 Socket Programming

This project also explored socket programming. This project used (TCP) sockets to accomplish all inter-node communication. While using an existing RPC library may have saved more time, the socket solution proved to be an interesting side-project. What made it so interesting were the considerations which had to be made when determining how best to use sockets. In broad terms, I identified three approaches, each with their own advantages and drawbacks:

1) Establish $O(n^2)$ sockets to tie each node together, reuse them.
2) Establish a new socket for every message issued, close it afterward.
3) Establish a pool of sockets on every node, reuse them when possible.

Approach (1) has the advantage of only opening and closing sockets when the connection is being established for the first time, or after the connection has been dropped. However, if one side of the connection has to reset (without dropping the connection) because it received unexpected data, then the other side cannot be sure what data (if any) it will receive next time it listens to the socket. In essence, there can be no such thing as a non-graceful teardown with approach (1).

Approach (2) solves this shortcoming by creating new sockets every time a message is passed between nodes. That way if one side needs to reset, the socket can simply be abandoned. The issue with approach (2) is that it is wasteful. Because the leading node must periodically send out heartbeats to its followers, approach (2) could easily require creating and closing $n - 1$ sockets (where n is the number of nodes in the system) once every tenth of a second.

Approach (3) takes the frugality of approach (1) and blends it with the cleanliness of approach (2). Each node holds two structures, a `ConnectionPool` for sending messages, and a `TerminalPool` for handling incoming messages. Both pools hold sockets. When a node tries to send a message to another node, the source node's `ConnectionPool` searches itself for an idle socket to the destination node. If one is found, it is extracted from the pool and passed to a function which creates a message. If one is not found, a new socket is created by connecting to the destination's `TerminalPool` before being passed to the same message-creating function. Depending on how many messages are being sent or received concurrently, there may be multiple sockets in a pool connected to the same node. If the message-creating function (or the message-handling function on the other end) must be terminated non-gracefully, then the associated socket is dropped from the pool. With this approach, one can reuse sockets on which previous communication terminated gracefully, while simultaneously being sure that no data from other messages will arrive on the socket while it is in use.

4 Stepping Through an Operation

To give a better impression of how the consensus framework works, this section explores how operations on the distributed data structure are performed. Figure 3 illustrates how a request to perform an operation (OP) is processed by the framework.

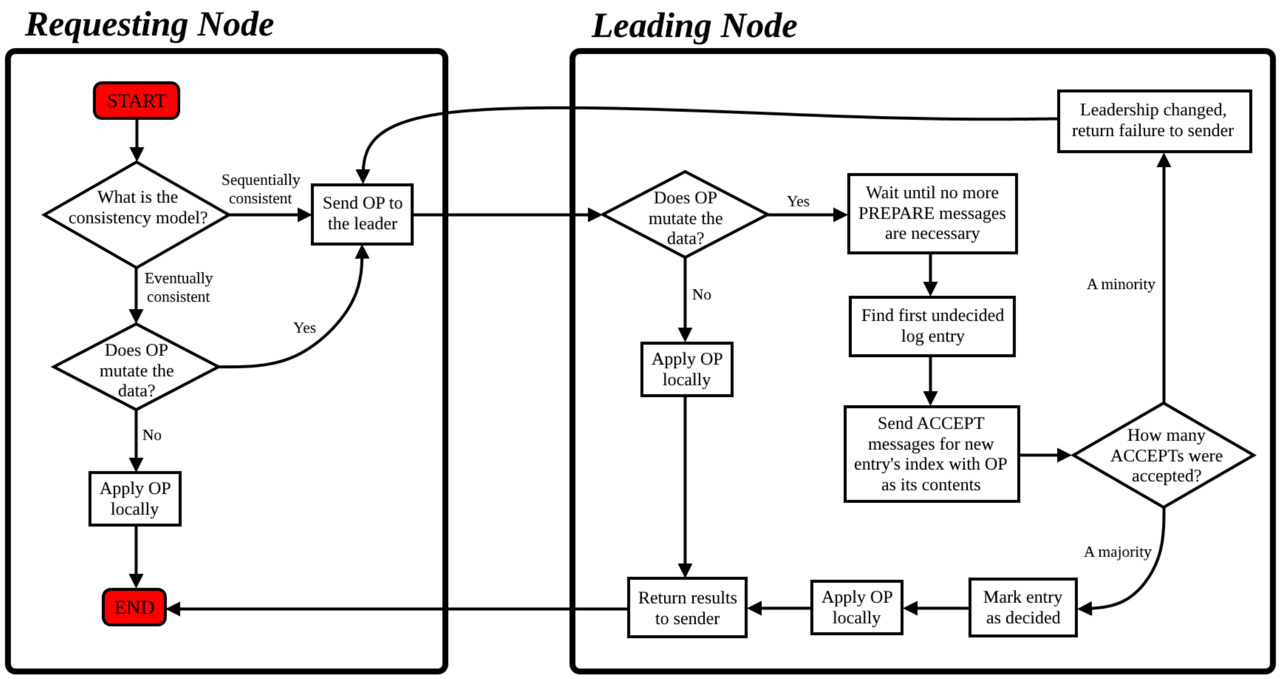


Figure 3 – The life cycle of a request to perform an operation.

However, it is important to keep in mind that (at the time of writing) not all of the processes described in Figure 3 have been fully implemented. In attempting to address other problem domains, I have been less able to spend time implementing the consensus portion of this project. In particular, the framework cannot yet issue PREPARE or ACCEPT messages, nor can it apply entries in the log. While all of the necessary abstractions are fully implemented (the list of valid operations, the log entries, the return packages, and the communication framework), the actual consensus module is still under construction. Despite its lack of completeness, I have still spent much time reasoning about how the consensus portion of this project should function.

The first thing that must be addressed before any operations can be performed is what operations should be logged? If the set of valid operations is considered to be all (visible) member functions of the type being distributed, then operations qualified `const` or `immutable` will never mutate the data being distributed. If applying these functions to the data structure will not mutate it, then these operations can be safely ignored by the log. The log only needs to store operations which change the state of the data being distributed. That way, the data structure can be reconstructed on other nodes by applying entries.

Next, one must consider what consistency model is in use. The consistency model in use will dictate the nodes on which non-mutating operations can be performed safely. For operations that must be logged, the consistency model does not matter. These operations must be sent to the leader so that they can be added to the log. In an eventually consistent setup, non-mutating operations can simply be performed locally. One consequence of eventual consistency is that a non-mutating operation is not necessarily guaranteed to see the result of a mutating operation which happened prior to it. This is because that mutating operation may not have been applied on the requesting node by the time the non-mutating operation is executed. Eventual consistency has the advantage of being highly available for non-mutating operations, but at the cost of appearing inconsistent until the leader can bring other nodes' logs up to date. On the other hand, with sequential consistency, all operations must be performed by the leader. Although non-mutating operations are not added to the log, sending them to the leader ensures that they will always be performed on a node where previously-requested mutating operations have been applied. Therefore, all operations which are causally related will appear ordered. Furthermore, this single order will be observed on all nodes, because all of them are sending their operations to the leader.

While applying a non-mutating operation is relatively easy, mutating operations require a little more work. Before any such operations can be considered, the leader must first wait until it is up to date with other nodes. This means waiting until no more PREPARE messages are necessary. At that point, an open log slot will be chosen, and ACCEPT messages will be issued for the new operation in that slot. If a majority of the ACCEPT messages were accepted, then the log entry is marked chosen, and the operation is applied to a local instance of the type being distributed. The results are then sent back to the node that requested to perform the operation. Otherwise, if a majority of ACCEPT messages were rejected, then leadership has changed. The requester is notified of the change, and the process begins anew with another leader.

5 The Limitations of the Implementation

While the aim of this project was the implement a consensus framework that would distribute *any* type, some limitations did make themselves apparent. One such limitation is that the valid operations on a distributed data type are limited to the accessible member functions of the type being distributed. Non-member functions which take the type being distributed as an argument are not considered valid operations on the distributed type. This is the result of a design decision made early in the project. Before I learned about D's reflection capabilities, I considered having the user specify what operations were valid for some type. This would have solved this issue. However, I realized that the list of valid operations would then have to be agreed upon by every node in the system. After learning about reflection, I decided to limit the list of valid operations to only include the member functions of the type being distributed (and generic get and set operations if the type lacked member functions).

Function templates are another one of the implementation's limitations. Because templates only generate code once they are supplied with arguments, reflection does not actually reveal any of the function signatures created when the template is instantiated. As a result, it is not possible to include function templates in the list of valid operations on a type.

Finally, functions qualified `inout` also impose limitations on the framework. The `inout` qualifier acts like wildcard with respect to mutability. Depending on the mutability of the arguments supplied to a function with `inout` parameters, the `inout` qualifier might resolve to any one of `const`, `immutable`, or mutable (which has no explicit qualifier). Therefore, if the function is marked `inout` (meaning the `this` parameter is qualified `inout`), then one cannot be sure whether or not the function will mutate the data structure being distributed. Reflection cannot solve this issue either. Consequently, all calls to operations marked `inout` must be logged, even in situations where `inout` would resolve to `const` or `immutable`. The only exception to this rule is when a function is marked `const inout`. Such functions will either resolve to `const` or `immutable`, but never mutable.

6 Conclusion

This project substantially expanded in scope as it developed. What initially started as a consensus problem quickly turned into a meta-programming problem with the features of a consensus problem. This was a consequence of trying to make the framework accept any arbitrary type. Had the aim of this project been to distribute some particular type, meta-programming would never have been a concern. Of course, consensus issues still played an important role. Because Lamport's description of Multi-Paxos does not recommend any particular means of leader election, the issue of how to elect a leader was one such consensus issue broached by this project. Although not all of the consensus logic is currently complete, problems associated with log replication were still explored in depth. Other issues like those related to inter-node communication were resolved more completely. Finally, though the framework's goal was to distribute any type, some limitations related to templates and type qualifiers still arose. This project, though it is not yet fully complete, demonstrates that it is possible to implement a type-independent consensus framework in the D programming language. It simply requires that one address issues that reside outside the domain of distributed consensus.

Though the course is now over, I plan to continue working on this project so that I may more fully explore its potential nuances.

7 Sources

[1] Leslie Lamport, “Paxos Made Simple,”
https://lamport.azurewebsites.net/pubs/paxos-simple.pdf (accessed April 3, 2020).

[2] John Ousterhout and Diego Ongaro, “Paxos Lecture (Raft User Study),”
https://www.youtube.com/watch?v=JEpsBg0AO6o (accessed April 3, 2020).