CSC 499 (Winter 2020) – Status Report Two

Michael Windels
February 21, 2020
V00854091

Abstract

This report summarizes the developments I have made on this project in the last three weeks. The first section covers what parts of the project are complete, like the interface layer of the distributed type. While the second section covers what issues I have been hashing out, but which are not yet implemented, like consistency guarantees, data isolation, and exception throwing.

1 Completed: The Interface Layer

Over the last three weeks, I have been implementing the interface layer of the distributed data type. As it stands, the interface layer is essentially complete. The distributed type is currently able to mimic the member functions, member variables, and types defined within other data types. It does so by reflecting on the members of some type `T` (where `T` is the type being distributed) at compile time and reconstructing their declarations as faithfully as possible. However, member declarations in the distributed type cannot always perfectly mimic their corresponding members in `T`. In particular, the `pure`, `nothrow`, `@trusted`, `@safe`, and `@nogc` attributes of member functions in `T` are not copied over to the distributed type. These attributes are not supported because a function with any one of these attributes cannot call a function that lacks those attributes. Currently, I do not think that the consensus logic will be able to assure any of these properties, thus I am ignoring them. For similar reasons, some currently supported function and parameter attributes like `ref` may lose support in the future. For an idea of what the interface layer of the distributed type looks like given some example type, see Figure 1. Also notice that members not marked `public` or `export` are not mimicked in the distributed type.

Original Type

```
struct Ex
{
    int a;
    const float b;
    immutable bool c;

    @("uda", 1) void f(immutable int i) immutable @nogc nothrow {}
    static ref float g(ref float x) {return x;}
    export final ref inout(int) h() inout {return a;}

    enum P { ... }
    struct Q { ... }
    union R { ... }
    private class S { ... }
}
```

Distributed Mimicry

```
class Distributed(T : Ex)
{
    public Distributed!int a;
    public const(Distributed!float) b;
    public immutable(Distributed!bool) c;

    public @("uda", 1) @system immutable void f(immutable(int) i);
    public static ref @system float g(ref float x);
    export final ref @system inout inout(int) h();

    alias P = Ex.P;
    alias Q = Ex.Q;
    alias R = Ex.R;
}
```

Figure 1 – The interface of the distributed type given some example type.

2 In Progress: The Consensus Layer

Since the distributed type's interface layer is essentially complete, the next step is to begin reasoning about and implementing the consensus layer. These sections will establish the direction the project will take for (at least) the next three weeks.

2.1 Multi-Paxos and Consistency

The consensus layer's primary purpose is to ensure a consistent view of the distributed data structure across the cluster of nodes. By default, multi-paxos should at least ensure sequentially consistent writes. This means that for any node in contact with the leader (because multi-paxos involves leader election), one will always observe a single total ordering of writes. The sequential consistency of multi-paxos is a product of the log entries being decided by quorum.

Though writes adhere to a single total ordering, reads may not necessarily observe this total order. Because reads do not alter state, reads do not have to be logged. Thus, reads could simply be performed locally without needing to notify the leader. In this case, however, a client may not observe writes which they performed prior to their reads. This could be because the results of their writes have yet to propagate back to the node which they are reading from. To remedy this, the consensus layer could linearize all reads and writes on individual nodes by forcing every operation they process through a queue. So long as clients always read and write on the same node, this would at least ensure that they see their own writes.

Using a queue to make reads consistent is only one approach. There are many consistency models to choose from. Ideally, the distributed type would allow the user to select the consistency model which best suits their needs. However, to keep this project manageable in the time that is left, I will likely limit the consistency model(s) to those which are easiest to implement or easiest to test for correctness with.

2.2 Safely Mutating the Data Structure

Once the nodes of a cluster learn that some log entry has been chosen, they will need to apply that entry to their local copies of the distributed data. This will necessarily have to take place in the background, because the node must be able to process incoming operation requests at the same time. Thus, I propose that each instance of a distributed data type should spawn a thread whose job it is to apply changes to the local copy of the data structure. If this entry-applying thread is also designed to process non-mutating operations (i.e. using the queue approach described in section 2.1), then this yields two useful side-effects. First, the data structure being distributed need not be thread safe. If only one thread is handling operations on the data structure, then there is no risk of operations accessing state concurrently. Second, it ensures that (non-shared) static members cannot be mutated in any way which would bypass the consensus layer (i.e. from another thread). This is because D guarantees that all (non-shared) static members are stored in thread-local memory.

However, the second guarantee raises another issue. Although non-shared static members cannot be mutated from other threads, they also cannot be accessed from other threads. This essentially strips all static members of their static-ness. To resolve this I propose that whenever a distributed type is instantiated for the first time, it should spawn a thread which guards only the access and mutation of static data. In every other way this thread would be analogous to the threads that guard instance data. To ensure that only one of these threads is created per type and per node, an atomic thread ID could be stored as a (private) static member of the distributed type. Every time a new instance of the distributed type is created it would atomically read the value then atomically compare and swap (if the read returned null) to create the static handler thread. Figure 2 demonstrates what this might look like using the `Ex` type from Figure 1.

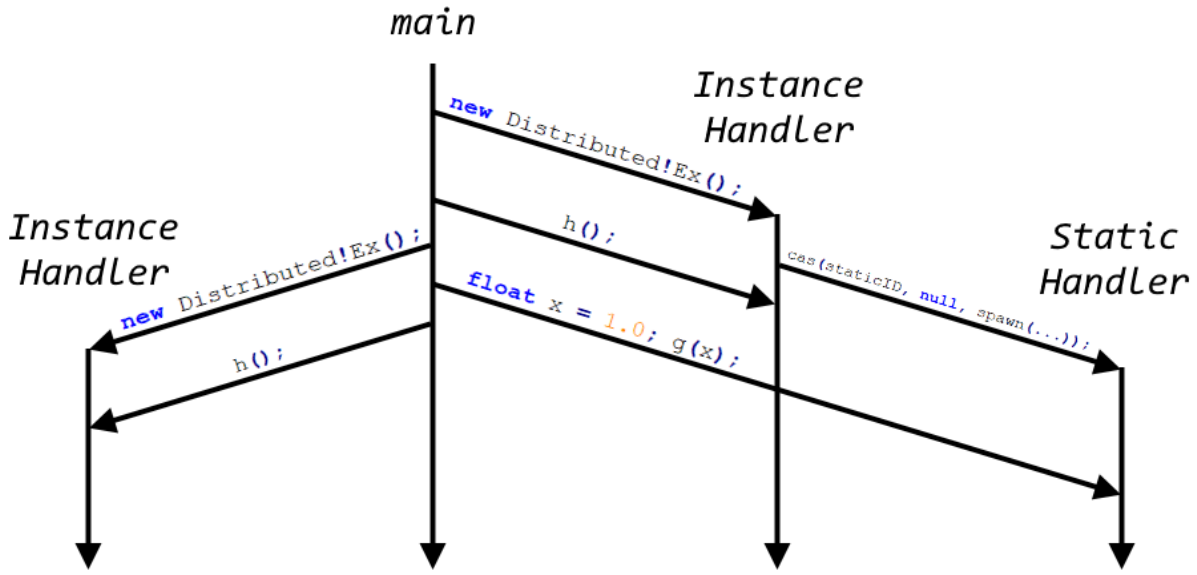

Figure 2 – The threads which handle the access and mutation of the `Ex` type being distributed.

2.3 Throwing Exceptions Over the Wire

Given that this project aims to distribute arbitrary data structures, some thought needs to be given to how exceptions should be handled. If a node proposes a command to the leader which throws an exception when it is applied to the data structure, that exception should be propagated back to whoever proposed that command. A straightforward solution would be to package that exception in with the operation's return values. Such a return package could include elements like:

- The operation's return value.
- The (mutable) reference and pointer arguments passed to the operation.
- Any exceptions thrown while applying the operation.

However, in order to deserialize an exception, its type must be known. Without knowing its actual type, the best one can do is to deserialize it as a generic `Exception` super-type. This is insufficient though, as it obfuscates type information that the user might rely upon. Currently I do not have a solution to this problem, but I will continue trying to solve it moving forward.

2.4 Miscellaneous Multi-Paxos Issues

In addition to the issues with consistency explored in section 2.1, I can foresee two other issues with multi-paxos. First is leader election. Leader election in multi-paxos is an optimization. Naive multi-paxos uses an instance of paxos for each log entry. With leader election, it is possible to eliminate most “prepare” messages. The issue with leader election is that there are many different ways to do it. For the purpose of this project, I will likely stick to a simple leader election scheme where the leader is always the node with the highest ID in the quorum.

The second issue with multi-paxos is variable cluster size. Permitting the cluster size to change while the cluster is running requires that additional constraints be imposed on the log. If time permits, I may explore variable cluster sizes. For now though, I plan to keep the cluster sizes static.