CSC 499 (Winter 2020) – Status Report One

Michael Windels
January 30, 2020
V00854091

Abstract

Since submitting the Pro Forma document, I've sketched out the structure of the distributed data type and devised a robust and portable format for the log entries. Some of the implementation details have been explored (i.e. compile-time reflection and mixins), but most of the implementation is upcoming work.

1 What's Been Going On?

I've started this project by sketching out a design for the distributed data type. During this process certain considerations have come to the fore, including how to structure log entries and how best to invoke the operations stored within them.

1.1 Structure of the Distributed Data Type

At its most abstract, the structure of the distributed data type looks like a three-layered cake. The topmost layer is the user-facing interface layer. This layer mimics the member functions and member variables of the original non-distributed data type. The member functions on the top layer make calls to the layer below; the consensus layer. This intermediate layer handles all of the consensus logic associated with multi-paxos and it contains the log. Below that layer is the final layer containing an instance of the original non-distributed data structure. As the consensus layer learns that log entries have been chosen, the appropriate function calls are made on the final data layer, updating the local copy of the distributed data structure. Figure 1 illustrates what the distributed data type looks like.

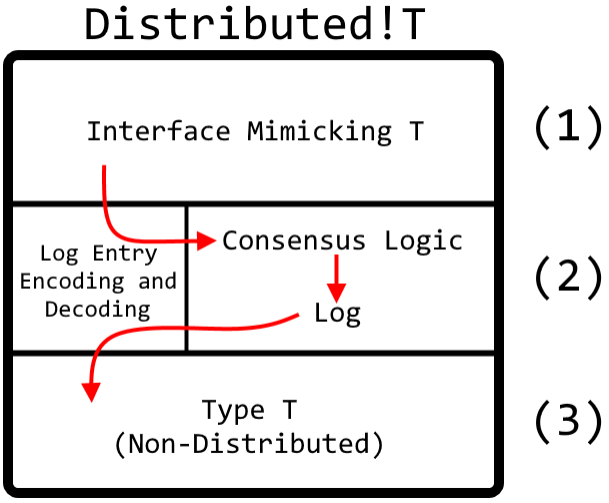
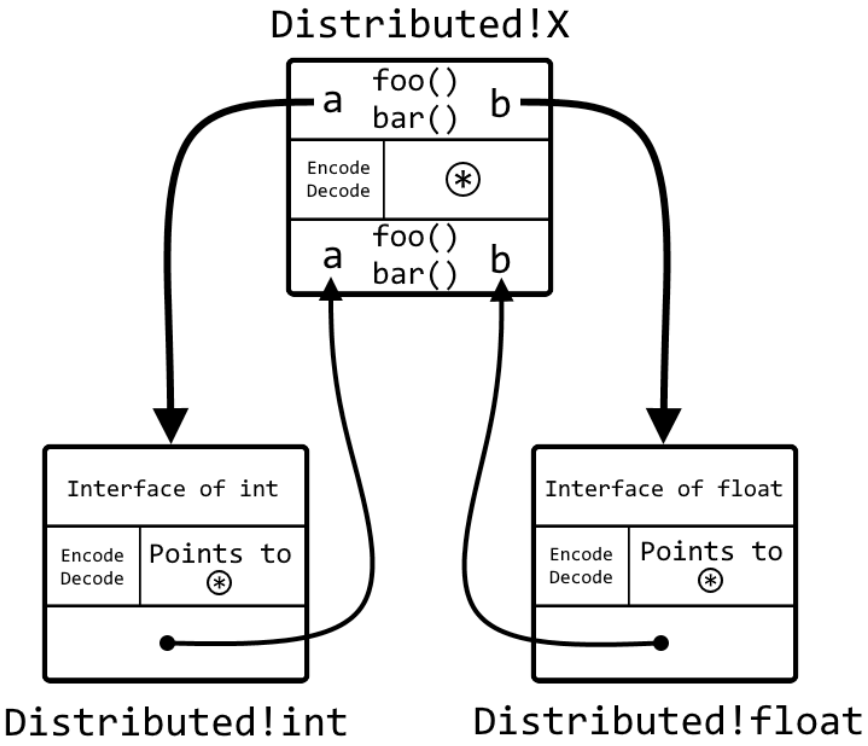


Figure 1 – The distributed data type and its three layers.
The red arrows demonstrate control flow for member function calls.

In order to accommodate the access and mutation of member variables directly (not through member functions), I believe that the distributed data type should have a tree-like structure. Each data member of the original (non-distributed) data structure will be wrapped in a distributed wrapper just like the parent structure. These child structures will be accessible through the top layer, but the log portions of their consensus layers will instead point to the consensus layer at the root of the tree. Figure 2 demonstrates what this tree-like structure might look like for some type *X*.



Where X is:

```
struct X{
    void foo() {}
    int bar() {return 0;}

    int a;
    float b;
}
```

Figure 2 – The tree-like structure of the distributed type for type X.

1.2 Structure of the Log Entries

Because the distributed data type aims to automate the process of distributing *any* arbitrary data structure, log entries need to be able to represent calls to the arbitrary structure's member functions. To accomplish this, I originally planned to simply write closures to the log. However, this presents two problems. First, closures may not necessarily be portable across nodes, especially because each machine will have an independent instance of the distributed data structure. A closure with bindings from one instance may not be compatible with another instance. Second, and more substantial, recording the bindings associated with a closure could necessitate writing a lot of data to the log. These bindings could even include snapshots of the entire data structure, defeating the purpose of keeping a log of operations in the first place. Rather than write closures to the log, I have instead decided to construct log entries as triplets containing the following information:

- A string representing the function's name.
- A tuple of function parameter types.
- A tuple of arguments supplied to the function.

The function's name and parameter type tuple help distinguish what function call to make (especially when overloads exist), and what types the arguments should be treated as when they are decoded. The argument tuple exists to be supplied to the function call made when the consensus layer wants to apply the log entry. A fourth element may also be included to help qualify the function's position in the tree illustrated in Figure 2.

1.3 Reflecting on Reflection

In order to automate the construction of the distributed type, the distributed type will need to perform reflection on its non-distributed base type. Reflection will permit the distributed type to obtain information about the member functions and member variables of its non-distributed base. D supports such reflection at compile time. D also supports what it calls the “mixin.” The mixin permits code injection (like eval in other languages), but its use is limited to compile time. Combined with reflection and template meta-programming, the mixin becomes especially powerful. With these three tools, the distributed type will be able to (1) mimic the interface of its non-distributed base, and (2) make calls to the non-distributed base structure using only the information stored in the log entry triplets.

However, compile-time reflection possesses a few limitations. First, D's compile-time reflection utilities do not recognize the types of a data structure's template members. Currently, my plan is to have the distributed type simply ignore template members of its non-distributed base. Second, as far as I can tell, D does not have a straightforward way to determine whether a member of a structure is a variable (as opposed to a function or template member). However, it may be possible to tell by process of elimination. Finally, reflection does not help solve the issue of static members in the non-distributed base. If the base structure contains mutable/mutating static members, then these could potentially be used to bypass the distributed type and directly modify parts of the underlying non-distributed base. In spite of these limitations, I still believe that compile-time reflection is the best tool available to construct the distributed type.

2 What's Coming Next?

Now that much of the shape of the distributed type has been mapped out, the next step is to actually implement it. Doing so will also necessitate delving deeper into multi-paxos, which I plan to do over the next few weeks as the implementation develops.