CPSC 426 Final Project: Hoplite Implementation in Go

Julian Lee
Yale University

Brian Choi
Yale University

Abstract

Task-based distributed systems are an approach for executing computationally demanding tasks, with popular applications in fields such as machine learning. In order to properly reap the benefits of parallelization and resource utilization offered by using these task-based distributed frameworks, an efficient collective communication layer is needed. Hoplite is a collective communication layer designed for task-based distributed systems with the unique contribution of a flexible communication schedule that can be computed in runtime. In this project, we implement the core features of the Hoplite backend using the Go programming language.

1 Introduction

For our project, we seek to implement the Hoplite backend, focusing on asynchronous and dynamic collective communication. [2] For specific function headers and gRPC details used in our implementation, see a more detailed specification here.

2 Hoplite's Workflow

In the execution process of a task-based distributed system, the application is split into tasks, where tasks are assigned to a specific worker node with an partial ordering for execution that needs to be enforced (e.g. if task 2 relies on the output of task 1). After the assignment of tasks to nodes through a task scheduler, Hoplite facilitates the communication setup and data transfer of objects between the worker nodes. For example, if worker nodes 1 and 2 execute tasks 1 and 2 respectively and task 2 relies on the output of task 1, worker node 2 will need to fetch the output object of task 1 from worker node 1 in order to execute task 2. Each node maintains a local object store containing objects it has used or computed in the past (e.g. past task arguments and outputs), and each object has an associated object ID. When a node receives a task along with its arguments in the form of object IDs, if the node doesn't store an object corresponding to one of these object IDs locally, it must retrieve this object from another node. This retrieval process involves two steps: 1) the node first searches the Object Directory Service (ODS) with the target object ID as a query to find a worker node (node A) that stores the corresponding object 2) the node sends an RPC call to node A requesting for the desired object to be broadcasted. The ODS is implemented as a sharded key value data store, where the ID of the object maps to the set of node locations that serve as a host for that object.

Hoplite's ODS provides a few benefits. First, the ODS is sharded across worker nodes, so each worker asks as both a client and server to this service; this allows the ODS to run without requiring additional servers. In addition, the coordination of the worker nodes with the ODS reduces the number of RPC calls. Rather than pinging every worker node in search for a desired object, the worker node now only needs to send two RPC calls: one RPC call to the ODS (in particular the worker node containing the shard with the desired objectId) and one RPC call to a worker node containing the desired object based on the ODS response. One key focus in our implementation was to replicate this contribution.

3 Implementation Details

Each worker node has three components: the ODS that is sharded across nodes, a local object directory, and a worker component that can execute tasks. RPC calls associated with each component are in Table 1.

Name	Description
ODS (requests sent to node with shard containing objectId)	
OdsGet	Find which node(s) host this object id
OdsSet	Inform the ODS that an object is hosted on a new node
OdsDelete	Inform the ODS that an object has been deleted from a node
Local Object Directory (requests set to node containing object)	
BroadcastObj	Sends specified object from local object store
DeleteObj	Deletes specified object from local object store
Worker Node (requests from task scheduler sent to any node)	
ScheduleTask	Schedules a task for execution on a node
GetTaskAns	Gets the output of a previously-scheduled task
DeleteGlobalObj	deletes a no-longer-useful object from all nodes.

Table 1: RPC calls for (worker) nodes in Hoplite. For more details, see the specification linked in section 1.

3.1 Object Directory Service

The object directory service works as a sharded key value data store, allowing us to build on our implementation for lab 4 in order to replicate the necessary functionaltiy. The data store maps the object IDs to their location information, which is the set of nodes that host the object in their local object store. When a specific object is requested using its ID, the object directory service returns a list of nodes that host a copy of the object. As discussed in Section 2, each worker node functions as both a client and server to the ODS. The ODS consists of the typical get, set, and delete queries. Anytime a worker node adds an object to its local object store, it sends a set request to the ODS to inform the service that it now possesses this object. When a worker node needs an object that is not in its local store, it sends a get request to the ODS. When a result is no longer used, the task manager could tell the worker nodes to delete a specific object, at which point the worker node would delete the object from its local object store and send a delete request to the ODS.

3.2 Object Management

When a Hoplite worker wishes to retrieve an object with a given objectId, it first searches its local directory, and if the object can't be found, it then finds a host node using *OdsGet* and retreives the object using *BroadcastObj*.

Because the Go programming language is strong and statically typed, in order to handle objects of different types we chose to serialize objects into bytes for storage in the local object store. The overhead of determining the proper type of the object and the involved serialization/deserialization is cast as the responsibility of the specific task. The details of the tasks we chose to implement and the types involved are elaborated on in further detail in the Testing section.

3.3 Worker for Tasks

The *ScheduleTask*, *GetTaskAns*, and *DeleteGlobalObject* RPC calls are the front-facing RPC calls of the worker node: these calls are used by the task scheduler. The worker for Hoplite offers both synchronous and asynchronous querying. The *ScheduleTask* RPC is an asynchronous query that returns an objectId as a promise before the task is complete. The *GetTaskAns* RPC is a synchronous query that returns the object associated with the specified objectId once the task is complete.

3.4 Task Scheduler

We chose to implement a simple task scheduler for demonstration and benchmarking purposes. Our task scheduler has two APIs: *ScheduleTask* and *RetrieveObject*. It essentially selects a node and then sends this node a *ScheduleTask* or *GetTaskAns* rpc call. Rather than choosing a random node, the task scheduler keeps track of how many requests are still outgoing for each node, and it chooses the node with the least number of outgoing requests.

4 Design and Tradeoffs

We will first talk about the trade-offs involved with using a sharded ODS to store the location of objects. An alternative approach could be to store the objects directly (as opposed to their location) in the ODS. This has the advantage of reducing the number of RPC calls required for a worker node to find an object that is not in its local object store: rather than making one call to the ODS and another call to the node containing the object, it can just make one call to the ODS. However, this comes at the cost of sending the entire object via RPC to the ODS every time a *OdsSet* request is made: if the object outputs are large (which is typical for many reinforcement learning workflows), this method of storing objects directly in the oDS becomes undesirable. Therefore, our ODS stores object locations as specified in Hoplite.

An additional design consideration with the ODS was whether to make the ODS run on a separate cluster of nodes. While running the ODS on the worker nodes themselves makes the system more conceptually complex (e.g. worker nodes function as workers that fulfill tasks, local object stores, ODS clients, and ODS servers), there is also benefits to this approach. Our approach makes better use of the computational resources available: when a worker node is not fulfilling ODS calls, it can work on tasks and vice versa. In contrast, a node with only ODS server functionality would likely be idle for the majority of the time.

A design decision for our task scheduling client interface involves offering asynchronous task scheduling and synchronous task output retrieval. When a task is scheduled, the user receives an objectId as a promise for the to-be-completed task. This allows for the scheduling of tasks that have dependencies on earlier currently-incomplete tasks. This is a desirable property for the user, since it allows them to conveniently schedule tasks concurrently in any order, and they only need to request to retrieve task outputs that they need. For example, if we have a series of tasks that eventually yield a single output O, the user can schedule all tasks in any order and then send a single request to retreive output O. All intermediary calculations/outputs remain in the worker nodes and are not exposed to the client or the user, simplifying client logic.

Another design consideration involves communication between the task scheduling client and worker nodes about the tasks themselves. Our worker nodes currently hold a predefined number of programmed tasks, and the task scheduling client passes a task id to select a task it wishes to be executed. For increased flexibility, the task itself could be passed in as a function to the RPC call schedule task: this would allow the worker nodes to execute any task as opposed to only those have have been explicitly defined. This is likely unnecessary since a fixed number of desired tasks can usually be predefined in a distributed task-based system. In addition, in most distributed task-based systems, the same task would need be run many times, making it inefficient to communicate this task through RPC for each task execution. A future optimization that has the efficiency of our system with increased flexibility could involve maintaining our task id framework, and adding another option to send in a custom tast via RPC if desired.

As a final small note on design, we needed to allow the flexibility for the user to also manually provide objects as task arguments as opposed to only being able to use objectIds associated with the output of a previous task. During the scheduleTask RPC call, the user can create custom objects with an unused object ID and send these to the worker node as task arguments via an objectId to byte array map; the worker node will first search this map for an object before checking its local object store and the global

ODS.

5 Evaluation

We implement 3 tasks as follows (input objectIds for any task can be promises):

1. Task 1: takes in one object ID corresponding to a list of (large) integers. Removes non-prime numbers from this list and outputs the updated list.
2. Task 2: takes in two object IDs, each corresponding to a list of (large integers). Element-wise multiplies the two lists together, removing extraneous element if list lengths don't match.
3. Task 3: given an argument list of objects to reduce, where each argument corresponds to an object storing an integer array, reduces objects to a single object through element-wise multiplication (generalization of Task 2).

Note that the reduce task can involve n objects that can be processed in any order. The reduce task streams in objects concurrently and reduces each object once it becomes available. Therefore, if one promised object takes much longer to become available than the others, this task can perform useful work and reduce the remaining provided objects in the meantime.

5.1 Testing

Using combinations of the three tasks above, we designed tests for the full functionality of Hoplite both from the node level and from the task scheduler level. Our main testing design was based on concurrently scheduling a large number of tasks, where approximately half of the tasks (assigned Task 2 or 3) relied on object futures to be generated from the other half of tasks (assigned Task 1). Then, we concurrently request the results from each task and do a quick correctness verification. We then send delete requests for all the created objects, wait for half a second for the objects to be deleted, and check to make sure we can no longer retreive the objects using $GetTaskAns$. This main test evaluates Hoplite's ability to execute concurrent tasks including tasks with promises on a 1-5 node system, and it also highlights Hoplite's ease of use since all the tasks as well as the $GetTaskAns$ requests can be sent concurrently whether or not any given task has been processed yet.

5.2 Microbenchmarks

Building on top of the testing functionality, we added microbenchmarks in order to evaluate the performance of our implementation. We computed two main benchmarks on the node level, in a configuration involving five nodes and five shards. We designed one computationally expensive benchmark (Benchmark A) and one computationally trivial benchmark (Benchmark B).

For each benchmark, we assigned 12 tasks. Half of the tasks were delegated to be Task 1, while the other half were delegated to be Task 2 (that took as argument outputs from Task 1 as object futures). Each integer array composed for the tasks was of length from 500 to 600 (uniform random distribution) with also a random number of prime numbers inserted into the arrays. For Benchmark A, the prime number inserted was 4952019383323, while for Benchmark B, the prime number inserted was 5. After fetching all the answers, all objects were deleted (all local object stores were cleared). The results are shown below:

Name	Time	Memory Usage
Benchmark A	6.126 s/op	768352 bytes/op
Benchmark B	0.804 s/op	809236 bytes/op

Table 2: Hoplite Benchmarking Results

6 Future Work

The Hoplite paper discusses implementation of a Python frontend and integration with Ray, a task-based distributed framework, in order for more complete benchmarking and evaluation of Hoplite. We listed this as one of the stretch goals of our project, and we ended up determining it to be beyond the scope of this final project, but in the future we hope

to perform this integration to be able to present a complete view of Hoplite and its applications.

Additionally, in addition to fault tolerance, there are optimizations for performance described in the paper that we did not implement due to time constraints. There are three main optimizations that we initially considered but did not implement for this project due to various constraints. Firstly, for the object directory service, Hoplite specifies a cache for small objects in order to avoid the overhead of querying the ODS when possible. Although this optimization offers latency improvements due to avoiding the complexities of object transfer when possible, we did not target this optimization for our project due to the complexities involved in designing an efficient caching system. Secondly, Hoplite specifies an optimization in the case that an object only requires read-only access to an object in its local object store. Our implementation always requires that the object data is copied for the receiver task to use, in case that the data is modified during the task. However, in the case of read-only access, the receiver task can simply access the data through its local object store instead of having to copy it over. Thirdly, the Hoplite paper describes a tree-structured algorithm for the *Reduce* operation in order to avoid a bottleneck on one node. If the reducing of multiple nodes was assigned to one node to handle, the efficiency of the reduce operation will be bottlenecked by the network bandwidth of the single node that is delegated the *Reduce* task. However, this algorithm avoids this bottleneck by conducting intermediate *Reduce* operations at each node along the tree, so that these intermediate nodes can make progress even when not all nodes involved in the *Reduce* operation are ready. For our implementation, we settled for the simpler version of *Reduce* on one node instead of implementing the tree algorithm due to time constraints, because we determined that our Hoplite implementation could still demonstrate useful workflows without this more efficient *Reduce* implementation.

Given of the scope of a class final project, we focused on being able to demonstrate an immediate workflow and therefore these aforementioned features did not make it off of the drawing board for now, but in the future as we focus on improving our Hoplite implementation we hope to demonstrate all of these features.

7 Related Work

There are many other frameworks for optimizing data transfer, such as MapReduce. [1] MapReduce handles situations of generating and processing large datasets through distributed parallel execution. For efficiency, the input task is split into sections to be handled by various worker nodes. MapReduce uses a *Reduce* functionality in order to combine the data associated with a specific key, similar to Hoplite, although MapReduce does not use the tree-like Reduce algorithm that Hoplite applies. The aforementioned description highlights clear similarities in how both MapReduce and Hoplite split workloads and then combine results afterward. However, Hoplite claims to be the first effort for optimizing the collective communication layer for task-based distributed systems, separating it from prior data transfer optimizations like MapReduce.

Acknowledgments

We would like to thank Professor Richard Yang and Xiao Shi for their guidance this semester.

References

[1] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, January 2008.

[2] Siyuan Zhuang, Zhuohan Li, Danyang Zhuo, Stephanie Wang, Eric Liang, Robert Nishihara, Philipp Moritz, and Ion Stoica. Hoplite: Efficient and fault-tolerant collective communication for task-based distributed systems. 2021.