The University of Washington Bothell
CSS-600: Independent Research

Distributed Data Structures

Brian Luger
Professor: Dr. Munehiro Fukuda

15 December 2021

Introduction

The purpose of this paper is to provide a summary of the work performed over Fall quarter, 2021. This consisted of two primary objectives. The first was to support the use of the new GraphPlaces implementation through bug-fixes, documentation, and the refactoring of earlier TriangleCounting and RangeSearch applications built to utilize the GraphPlaces framework. The second, larger, objective was to implement a distributed map, built on the new Aeron[1] messaging framework, to replace the deprecated Hazelcast[2] distributed map. This paper will briefly speak to the application refactors before going into detail regarding the implementation of the distributed map.

Background

GraphPlaces is an extension of the existing MASS Places object that adds capabilities for constructing graphs of places objects with which simulations can be run. Vertices within the graph are distributed across all nodes as shown in Figure 1 below. Vertices are distributed in a round robin fashion to ensure they are balanced across the entire cluster, and are persisted in a single vector on each node. Each vertex object within the graph contains a list of all its out edges, effectively making the container a distributed adjacency list.

[1]https://github.com/real-logic/aeron
[2]https://hazelcast.com/

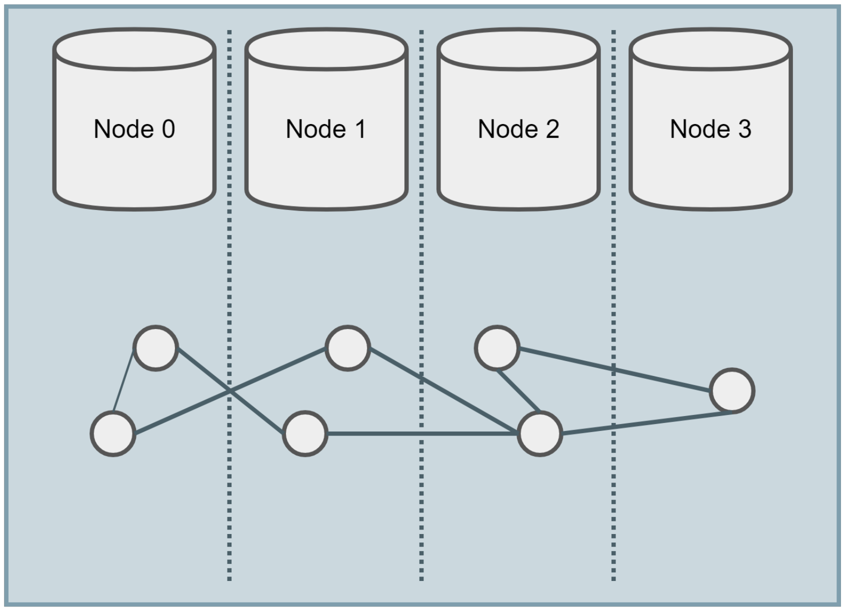

Figure 1: Visual representation of a distributed graph.

GraphPlaces utilizes a distributed map within MASS as a property store to optionally associate an object or label with the ID of its respective vertex. In doing so, developers can give names to vertices that better match the use case of their application, and perform lookups for a vertex using its names from any node within the MASS cluster.

Motivation

The impetus for this work stems primarily from the deprecation of the existing Hazelcast distributed map. The distributed systems lab at UW Bothell decided upon taking this action when making the decision to replace the Hazelcast messaging framework with Aeron. Because the distributed map is dependent on this underlying messaging system, it too needed to be replaced.

In addition to this, MASS did not have a distributed map of its own and there were a few bugs within the integration of the Hazelcast distributed map that made it a good opportunity to replace and/or refactor.

GraphPlaces Support

As mentioned above, support for GraphPlaces came in the way of implementing bug-fixes, writing documentation, and refactoring the TriangleCounting and RangeSearch applications within the `mass_java_appl`[3] repo.

[3]https://bitbucket.org/mass_application_developers/mass_java_appl

Triangle Counting

Triangle counting is a graph based algorithm for counting the number of triangles within a provided graph structure. In this context, a triangle is defined to be a grouping of three vertices such that all three connect as shown in figure 1 below.

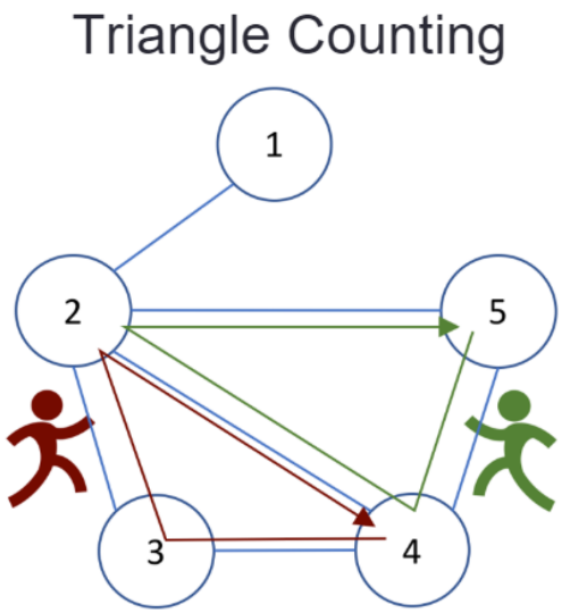

Figure 2: Visual depiction of triangle counting.

The graph structure is provided via a file in the DSL[4] format and is distributed across all nodes within the MASS cluster. In this case, the implementation had already been done and simply needed to be updated to ensure that it not only worked with the new GraphPlaces implementation, but could also run deterministically with multiple nodes in a cluster.

The updates to make this happen consisted of utilizing the new methods for importing graph files and ensuring that vertex IDs weren't being calculated by the application, rather they were owned by the GraphPlaces class. Additionally, the agents used in the implementation needed to be made to extend from the GraphAgent, as opposed to the standard MASS agent, so that their functionality worked appropriately over the graph structure.

Range Search

Range search is an algorithm often used to determine if a set of objects lies within a provided range. A good example of this could be a geographical map of a city with point objects representing the latitude and longitude coordinates of popular destinations. If a user wanted to know all the popular destinations within a specific area on the map, they could provide the algorithm with a range of latitude and longitude values and the algorithm would return to them all of the points that are contained within that range.

[4]Distributed Systems Lab graph file format

Its implementation within MASS makes use of a KD-Tree data structure built on top of the GraphPlaces framework. This tree is then populated with a list of 2-dimensional points that are loaded in from a file. When running the application the user provides a rectangular range to search for by providing minimum and maximum x and y values and the algorithm returns all of the points within that range.

Updating the RangeSearch application to work with the new GraphPlaces implemenation was quite a bit more effort than that of updating TriangleCounting. This was primarily do to some flaws in its original implementation, as well as how it utilized the previous GraphPlaces framework. Instead of updating the existing project, I felt a ground up re-write of it would be cleaner and more productive.

The re-write of this can be broken up into two primary tasks. The population of the KD-Tree, and the range search over said tree.

KD-Tree

A KD-Tree is a spatial data structure for organizing points in k-dimensional space within a binary search tree. Points are added to the tree such that the dimension used to determine their placement alternates based on the height of the tree. For example, the first level of the tree may place points based on the `x` value of a point, the second level the `y` value, third `x`, fourth `y`, etc...

In GraphPlaces, the creation of a node in this tree is equivalent to creating a vertex and adding the appropriate edges. To do this, a recursive function is used to traverse the tree and add a vertex with the appropriate point data based on the height and point value. An excerpt of this implementation can be found below:

```
...
// x-coord
if (level % this.dimensions == 0) {
    if (p.x <= root.value.x) {
        if (root.left != -1) {
            Vertex leftChild = getVertex(root.left);
            insertPoint(leftChild, root.left, p, level+1);
        } else {
            root.left = this.addVertexWithParams(p);
            this.addEdge(rootID, root.left, 0);
        }
    } else {
        if (root.right != -1) {
            Vertex rightChild = getVertex(root.right);
            insertPoint(rightChild, root.right, p, level+1);
        } else {
            root.right = this.addVertexWithParams(p);
            this.addEdge(rootID, root.right, 0);
        }
    }
    // Migrate agent to the root node and update its left/right fields.
    this.agent.callAll(UpdateAgent.update_, new int[]{root.left, root.right, rootID});
    this.agent.manageAll();
...
```

As can be seen on line 3 of the excerpt, the dimension of the point is used to determine its placement, and it varies based on the tree level. Line 22 shows the use of agents to update the left and right child nodes for each point added to the KD-Tree. At this time, GraphPlaces doesn't have a means of updating existing vertices within the graph. Agents are used to help mitigate this limitation however it comes at the expense of increased construction time for the tree.

This is because an agent needs to be created for each point and it must traverse a tree that is maintained in a distributed graph data structure, meaning it must hop between systems in the cluster. One potential way of alleviating this could be to add support for updating the state of existing vertices within the graph based on their ID. This would greatly reduce the time required to construct the KD-Tree as, at most, one round trip between systems would be needed to update it.

Range Search Algorithm

The range search algorithm itself is quite simple. It starts at the root node, or vertex, and recursively checks whether the region comprised of the nodes children lies within the query region provided by the user. If it does, it continues to traverse child nodes, recording those that fall within the range. If it does not, there is no reason to traverse that particular branch any further and thus it moves on to adjacent nodes. This enables the algorithm to discard large sections of the tree thus saving the significant amount of time that would be required to traverse them otherwise.

In GraphPlaces, the algorithm is the same. It recursively traverses the tree, dismissing sections that are outside the provided range and recording vertices that contain points within it. An excerpt of this can be seen below:

```
...
// If the point is within bounds add to results
if ((root.value.x >= minX && root.value.x <= maxX) &&
    (root.value.y >= minY && root.value.y <= maxY)) {
    results.add(root.value);
}

// x-coord
if (level % this.dimensions == 0) {
    // left-child
    if (root.value.x >= minX) {
        rangeSearch(results, getVertex(root.left), level + 1, minX, maxX, minY, maxY);
    }

    // right-child
    if (root.value.x <= maxX) {
        rangeSearch(results, getVertex(root.right), level + 1, minX, maxX, minY, maxY);
    }
}

// y-coord
...
```

Though this implementation doesn't utilize agents, one could be done that does. I opted not to for the following reasons. First, agents cannot traverse a graph recursively. It would need to be done in an iterative fashion using an auxiliary data structure to maintain state of where the agent has been. This data structure would grow and shrink based on the number of vertices in the graph and would need to be sent along with the agent as it migrates between systems in the cluster. As such, I'm not convinced it would scale well with the size of the graph. That said, with the introduction of a distributed map that can be used from any node in the cluster, it would be possible to maintain the queue or stack within the map. This would prevent the need to send it along with the agent, and potentially making it a viable approach.

The second approach I opted not to pursue was that of simply creating an agent for every vertex in the graph. In doing this, all agents that reside on a vertex containing a point within the provided range could simply report as such via a `callAll` function. However, this defeats the purpose of both the KD-Tree and the RangeSearch algorithm so it didn't seem like an appropriate solution to pursue.

Distributed Map

The implementation of the distributed map consists of three high level problems. The first is how the key/value pairs within the map are distributed across the cluster. The second is how to facilitate messaging between nodes in the cluster such that data can be requested by and sent from different nodes. This is needed to make it possible for function calls such as `get`, `entrySet`, `reverseLookup`, and other functions that require the retrieval of key/value pairs residing on remote systems, to work. Lastly, these messaging primitives need to be utilized in such a way that the function calls that require communication with remote nodes, block until they are able to return the appropriate values to the caller. Doing this correctly should also make it easier to implement asynchronous function calls in the near future.

Distribution of K/V Pairs

Distribution of key/value pairs is done through the use of consistent hashing. Consistent hashing is an algorithm used often in load balancing state across a distributed system. It makes use of a hash ring to determine which nodes have ownership over which key/value pairs.

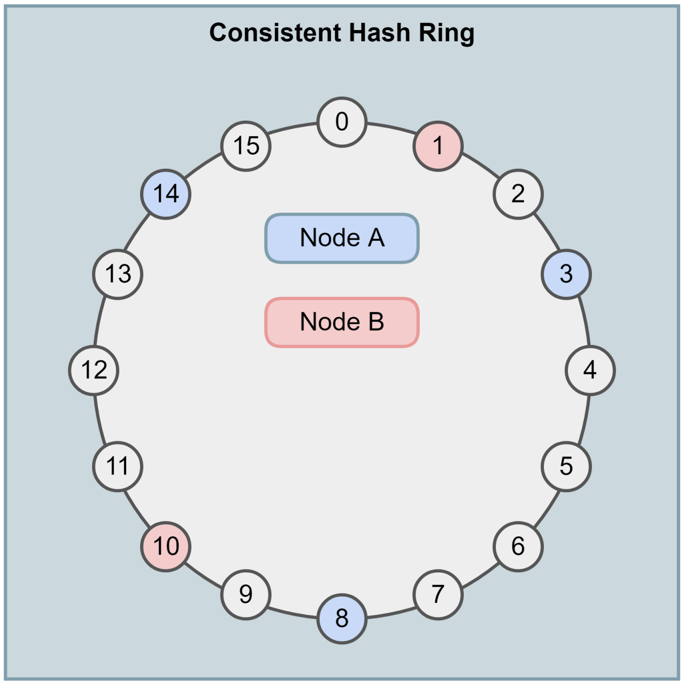

Figure 3: Consistent Hash Ring.

In figure 3 above, nodes A and B are hashed and placed on the ring, which is comprised of the set of integers. Key owners are then found by hashing the key to determine its position on the ring, and then locating the nearest node in a clockwise fashion. A collision resistant, uniform, hashing algorithm is used to hash the node ID and additional prefixes are added to give them multiple positions on the ring. The more positions given to each node, the more evenly distributed the key/value pairs will be. In MASS, we use the SHA256[5] hashing algorithm and a default of 20 positions per node. A TreeMap is then used to associate a node ID with its hash value, or position, on the ring. This allows us to keep nodes sorted based on their position while still enabling fast lookups and keeping code complexity low.

Node-to-Node Messaging

With distribution of key/values pairs handled, the system now knows where to put or get a key/value pair given the key from the caller. However, it still requires a means of communicating with that node to retrieve that value. In MASS, this is commonly done using the `MProcess.java` class. This, however, will not work for the distributed map as it needs to be usable from all nodes in the cluster. MProcess is strictly limited to sending requests from the primary node. To mitigate this, Matthew Sell[6] built support for node-to-node messaging into his Aeron-based messaging framework.

His work enabled the primitives needed to send arbitrary messages to nodes within the

[5]https://datatracker.ietf.org/doc/html/rfc6234
[6]http://depts.washington.edu/dslab/MASS/index.html

cluster in a reliable fashion over UDP[7]. However, for the distributed map to work, it still needed the ability to send a message to a node, and have the node return a response. This is needed to allow functions such as `get`, that require the distributed map to send the key to a remote node and receive the value associated with it.

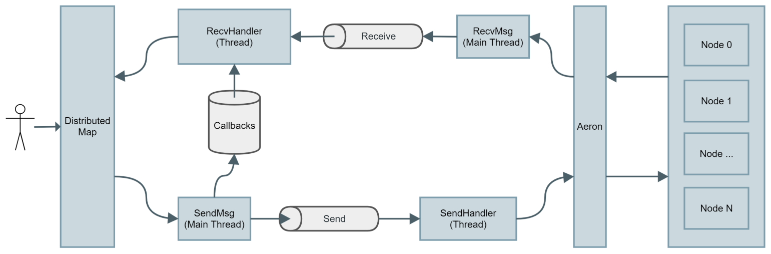

Figure 4: Request/Response messaging.

To enable this, a `NodeToNodeMessenger.java` class was introduced into the messaging package. This class uses concurrent queues and multiple threads to decouple the sending and receiving of messages from their processing. As seen in figure 4 above. When a function of the distributed map is called that requires communication with a remote node, for example a `get` call that provides a key and expects the associated value. A message is created and placed on the send queue. The function signature for doing this, looks like this:

```
...
public void sendRequest(int nodeID, S message, Consumer< S > callback)
...
```

The caller provides the message, the ID of the node to send it to, and a callback function that is executed when its response is received. Callback functions are stored within a ConcurrentHashMap and associated with a message ID. From here, a sendHandler running on a separate thread processes the messages by sending them to the destination node using the underlying Aeron node-to-node messaging framework.

When the other node receives the `get` function call message, it processes it using a requestHandler defined by the class that extends from NodeToNodeMessenger, in this case, the distributed map. This works in much the same way as MProcess does, using an enum to define function IDs and calling the one provided in the message. Below is excerpt of the MapMsg type:

[7]Uniform Datagram Protocol

```
...
public class MASSMapMsg implements Serializable {
    // Function is an enum that specifies the remote function being requested
    // by the MASSMapMsg. These effectively map to the public functions of the
    // distributed map.
    public static enum Function {
        ...
        GET,
        PUT,
        REMOVE,
        REVERSE_LOOKUP,
        ...
    }

    public Function func;
    public Object arg1;
    public Object arg2;
    public Object returnValue;
    public boolean success;
    public String errormsg;
...
```

The key difference between this and MProcess, aside from this being usable on all nodes in the cluster, is that the package wanting to make use of node-to-node messaging owns the message type, the functions that can be called with it, and their implementations. This is counter to MProcess in that all packages that wish to utilize it must couple logic to it.

When the node receives the request, it calls the appropriate function based on the content of the message and returns the value. In the case of our `get` call, it returns the value associated with the key. This is then sent in a similar fashion back to the calling node.

Once the calling node receives the response message, it is en-queued in the receive queue which is then processed by the receive handler running on a separate thread. This handler reads in the message and looks up the callback function associated with the originating message ID. If found, it calls the callback function, thus providing the original caller with the appropriate response.

Note that this is all done in an event driven manner without the need for wasting CPU cycles on busy waiting for messages. Both queues are `LinkedBlockingQueues` which allow the handlers to block until a message is available. When MASS is signaled to shutdown, the close method in the NodeToNodeMessenger is called, which interrupts the handler threads and forces them to return, thus shutting the messenger down gracefully.

Sending/Receiving

Once the ability to send a node a request message and be able to receive an associated response had been implemented on top of Aeron, the distributed map still needed to make use of it for messaging. To do this, the distributed map message, shown above, had to be created for sending and receiving map specific messages. Additionally, the distributed map needed to extend from the NodeToNodeMessenger and define a `requestHandler` function. This enables the request/response based messaging and provides the messenger with a function to call when it receives new map messages. This function also looks similar to that of MProcess in that it defines a switch statement for handling the various map related functions. The difference again being that the logic is decoupled from other packages within MASS. An excerpt containing a part of the `getRequestHandler` function can be seen below.

```java
...
@Override
public Function<MASSMapMsg, MASSMapMsg> getRequestHandler() {
    return ((MASSMapMsg msg) -> {
        // call the appropriate function and return the value if necessary.
        // get, put, etc...
        switch (msg.func) {
            case CLEAR:
                this.map.clear();
                msg.success = true;

                return msg;

            case CONTAINS_KEY:
                msg.returnValue = Boolean.valueOf(this.map.containsKey(msg.arg1));
                msg.success = true;

                return msg;

            case CONTAINS_VALUE:
                msg.returnValue = Boolean.valueOf(this.map.containsValue(msg.arg1));
                msg.success = true;

                return msg;
            ...
```

Because messaging is asynchronous in that function calls return immediately and provide return values via callback functions, utilizing them in the distributed map was found to be a bit more challenging. This stemmed from the need to provide the user with a way to call functions, such as `get`, and reference the return value in the response from within the same thread. The way messaging works is that the send function is called by the main thread but the callback function is called from the `receiveHandler` thread.

To accomplish this in an event driven manner, a CSP[8] approach was implemented. In this way, different threads utilize input/output channels for communication, as opposed to shared state[1]. In the map implementation, this was done using a concurrent queue. When, for instance, a `get` function is called, a queue is created and shared between the calling thread and the thread handling the return value. After the message is sent, the get function on the main thread, blocks until the queue is populated with the response, which is later en-queued by the receive handler thread. Once received, it is then returned to the caller. An excerpt of this implementation can be seen below:

[8]Communicating Sequential Processes

```
public value_type get(Object key) {
    ...
    // Otherwise, ask the node that should own the key for its value.
    LinkedBlockingQueue<MASSMapMsg> resultQueue = new LinkedBlockingQueue<MASSMapMsg>();
    this.sendRequest(
        owningNode,
        new MASSMapMsg(
            MASSMapMsg.Function.GET,
            key
        ),
        // callback function just adds the message to the result queue.
        // done by receiveHandler thread.
        (msg) -> {
            resultQueue.add(msg);
        }
    );

    // main thread blocks until it receives the results.
    return resultQueue.poll(this.RequestTimeoutInSeconds, TimeUnit.SECONDS).returnValue;
}
```

To prevent deadlock from occurring if a response fails to be received, a polling timeout is added such that once expired the thread will be interrupted and the call will return and log an appropriate error. This timeout is configurable in the constructor of the distributed map but is set by default to 10 seconds. All the remote distributed map functions were implemented in a similar manner to the `get` function.

An advantage of using CSP for communicating between threads in this use case is that it opens the door for allowing asynchronous function calls directly from the map. This could be a future enhancement and would allow the user to call a map function and have it return a Future immediately[9]. This would simplify concurrent programming from the callers perspective, allowing them to more easily overlap communication and remote workloads with that of those running on the main thread. A great example of this, using the distributed map, could be with reverse lookups as they can be an expensive operation. The user could call `reverseLookup` on the map, be handed a future, then continue with other processing on the main thread until they are ready to process the results of the reverse lookup call.

Conclusion

To conclude, the distributed map was tested with a control set of key value pairs and a MASS application that went through and checked to ensure all functions worked as intended on both single and multi-node deployments. In addition, it was tested with both the TriangleCounting and RangeSearch applications mentioned earlier in this report. While primarily used by GraphPlaces, I think the distributed map can provide value as a property store for other systems within MASS, such as Places, SpacePlaces, and Agents.

Additionally, the new node-to-node messaging framework enables MASS developers to implement messaging within their own feature packages as opposed to within MProcess. Thus preventing the need to modify MASS core to do so, as well as providing a tighter

[9]https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/Future.html

separation of concerns and less coupling within MASS. In fact, all of what's currently in MProcess could likely be cleaned up and moved into their respective packages.

With respect to future work, regarding the distributed map, the most notable addition would be asynchronous function calls from the map to make it easier to overlap calls with other logic in the main application thread. Secondly would be to benchmark the distributed map in terms of CPU and memory utilization. Because it's a distributed data structure, the expectation would be that execution time would take a little longer than that for a local map due to the round-trip latency between nodes. However, since we utilize consistent hashing, that latency should remain constant and not scale up with the number of system in the cluster. I would expect less memory usage between nodes as key/value pairs would be distributed uniformly. To what extent either of these occur, I'm not sure. It would be interesting to test.

References

[1] C. A. R. Hoare, “Communicating sequential processes,” *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, Aug. 1978. [Online]. Available: https://dl.acm.org/doi/10.1145/359576.359585