REYKJAVIK UNIVERSITY

T-419-CADP PROJECT REPORT

Overlay Networks

Kristófer Fannar Björnsson
kristoferb21@ru.is

Logi Sigurðarson
logis21@ru.is

Group 5

Teacher: Marcel Kyas

March 18, 2024

1 Introduction

This report explains the implementation details of our Overlay Network as well as answers some questions about it structure and behavior. The report's structure closely follows the questions asked in the rubric, but for further details on the implementation and how to run the code, we refer to the project's README.md file, as well as comments in the codebase.

2 Registry

2.1 ID Allocation

The node registry gets initialized with an slice of int32 ranging from 0-127. Each node gets a randomly selected index from 0 to the length of the slice. The value of that index becomes that nodes id, and then the value is removed from the list. Thus the id space continuously decreases while new node register. This ensures efficient id allocation, since each node needs only one try to get assigned an id.

2.2 Partition avoidance

Partition avoidance is achieved by sorting the slice of Ids before generating the routing tables and by using a circular buffer. Doing this ensured we get an overlay network that resembles a circle. This is because each node points to its respective neighbours depending on id.

2.3 Deadlock freedom

2.3.1 Registry

Our solution tried to avoid using locks as much as possible, thus lowering the probability of a deadlock. The registry has an instance of a mutex and is only used on core functionality in the registry; AddNode, RemoveNode and GenerateRoutingTables because these functions access the shared data structures of the Registry. In reality the shared data structures are only accessed by two goroutines, the MessageProcessing goroutine can call AddNode and RemoveNode, and the CommandLine goroutine can call GenerateRoutingTable. Because Add- and RemoveNode is only called before setup is called, and GenerateRoutingTable is only called after setup so in theory the locks for the core functions of the registry are redundant, but they are kept for the up most safety.

But because of this, a deadlock should be impossible since the gorouroutines should never try to access the shared data structures of the Registry at the same time. Additionally, neither goroutine tries to aquire another lock while holding a lock. Thus condition II for a deadlock, Hold and Wait, does not apply.

Some data structures in the Registry are only read and written to by the MessageProcessing goroutine and thus it was deemed that locks were unnecessary for these data structures.

2.3.2 Message nodes

The concurrency patterns used for the message nodes are:

- **Goroutines**
  As goroutines are used to create new *processes* which can concurrently run separately from other goroutines, on their own they can't cause any deadlocks, as a critical section isn't being shared here. However, with the existence of multiple goroutines, we'll have to consider how they communicate with each other.

- **Channels**
  While some goroutine share variables in a sense (one goroutine writes to it, another reads from it), the only explicit communication between the routines are the packet channel. This is represented in the Sender goroutine listening on the channel, while the CreatePackets & Listener goroutines send to it. The Sender can be halted if no packets exist on the channel, but that can not be classified as a deadlock, as the goroutine will be halted until there is something for it to do. It is supposed to halt at that time. On the other hand, the CreatePackets goroutine, as well as the goroutines created for each listener packet, can be halted if the channel buffer is full. But if this occurs, that means that the Sender goroutine is processing packets from the listener. In conclusion, out of the four conditions required for deadlocks, this situation doesn't fulfil the *Hold-and-Wait* condition, as no goroutine can wait while doing something that requires another goroutine to wait. The pushers and poppers of the channel being halted is an EXCLUSIVE OR condition.

As no explicit locks are used for the message side of the solution (fore every variable, only a single goroutine writes to it), that part of the code is free of deadlocks.

3 Nodes

3.1 Packet Delivery

Each node in the network has two main goroutines: A listener and a sender. Initially, the listener is setup to listen on a port, handling each incoming connection through a separate goroutine. Meanwhile, the sender, once it has received its routing table, connects to its neighbours and stores the connection in its modified version of a routing table.

The routing table, once received from the registry, is reformatted into a list of *ExternalNodes* (Table 1), sorted by the Id attribute.

Attribute	Type
Id	int32
Address	Address (Table 2)
Connection	net.Conn

Table 1: ExternalNode struct

Attribute	Type
Host	net.IP
Port	uint16

Table 2: Address struct

With this setup, all nodes are concurrently connected both as listeners and senders on the network. We have everything to start communications between them.

Once the nodes receive the InitiateTask message from the registry, all nodes use a new goroutine for creating packets. These packets are sent to a channel, which the sender continuously receives from. When a listener goroutine receives a NodeData packet from a connection, it verifies that the packet is valid, and then sends that packet to the channel. As we can't ensure that goroutines stacking packets on the channel won't be blocked when it fills up, and we believe the goroutine stacking the packet on the channel shouldn't be blocked from receiving new packets, we create a new goroutine exclusively for inserting onto the channel, which will finish once it manages to push the packet onto it.

Now, the sender goroutine pops from the channel using a range loop syntax[1]. For each received packet, the sender calculates the best neighbour and then sends the packet to that neighbour.

Our FindBestNeighbour algorithm essentially finds the neighbour with the nearest Id less than or equal to the destination Id. To calculate the best neighbour for a packet with destination x, we iterate through the routing table starting at the last index, as it has the highest Id (we have already sorted the routing table list), until we find a neighbour with an Id less than or equal to the destination Id ($Id <= x$). As soon as a candidate is found, its index is stored in a variable and the loop is broken.
If no candidate neighbour is found, that means that there is no neighbour with and Id less than or equal to the destination's Id, and so we select the last neighbour, the one with the highest Id, to be the candidate. This is because our nodes are connected in a circle, using modulus to connect neighbours near the 127 and 0 Id spaces. The neighbour with the highest Id is therefore closely connected to nodes with very small Ids.

This method ensures that no packet travels further than it needs to, as with each relay, it will reach a node closer to the destination node. Furthermore, if a listener goroutine receives a packet which has its Id marked as the source, it treats the packet as malformed and drops it. This result could be caused from a malformed packet, but also because of a packet travelling circles around the overlay network. Therefore, treating it as malformed is okay, but not without logging the event as a warning, which we haven't caught since we discovered a bug in our routing table code a long time ago.

Extra (on performance)...

> While we acknowledge that our routing algorithms isn't *blazingly fast*, having a time complexity of *O(n)*, it really doesn't matter that much, as the routing table configuration is setup to have very few entries (the logarithm of the amount of nodes in the system). Therefore, for a system containing a trillion (1,000,000,000,000) nodes, the routing table would only consist of about $\log_2 1,000,000,000,000 \approx 40$ entries. With such a large system, we have larger worries than how to improve an algorithm iterating over 40 items.

[1] As can be seen used on line 19 at https://go.dev/tour/concurrency/4

3.2 Duplication Avoidance

When calculating the best neighbour, there can only ever be single candidate, as a bestIndex variable is created and a pointer to that neighbour from the index is returned. The goroutine calling FindBestNeighbour, is the sender, which only calls the function on a packet which it receives from the packet channel. After the best neighbour is found, the packet is sent to that node and the packet is released. The packet will only ever once appear on the channel, either when the CreatePackets goroutine creates the packet and sends it on the channel, or when a listener goroutine has received a valid packet. This ensures that no packet is sent twice from the same node.

3.3 Task Completion

A specific goroutine is created at each node to check whether all packets originating from itself have been sent. Once a message node has finished sending all its packets, it prepares sending a TaskFinished message to the registry. This is quite hard to get exactly right, as there is no way for nodes to know whether they will have to relay packets later on. Therefore, our nodes are still able to relay packets even when having sent the TaskFinished message.

When the TaskFinished message is sent, that same goroutine expects a RequestTrafficSummary packet, after which it immediately responds with a TrafficSummary packet, containing all collected stats from throughout the node's existence and participation in the network.