Scalable Computing Systems Laboratory

École Polytechnique Fédérale de Lausanne

Design and implementation of a decentralized car sharing matching algorithm

by Jeremy Dezalos

Bachelor Project Report

Prof. Dr. Anne-Marie Kermarrec
Thesis Advisor

Rafael Pires
Thesis Supervisor

EPFL IC IINFCOM SaCS
BC Building, Office BC 347
1015 Lausanne / Switzerland

June 11, 2021

Contents

Chapter 1

Introduction

During the recent years, matching algorithms have taken a significant place in our lives. We use them for finding love, a job, an accommodation or just a ride. Most of those algorithms we interact with are held closely by a sole owner. All centralized algorithms share the same problem: if the server goes down, all the service become unavailable. Also, a user has no guarantee that the results it received are not biased by hidden factor put in place by the owner.

Our goal is to provide a fast, reliable and scalable solution that will answer to this problematic. We introduce a decentralized matching algorithm capable of delivering messages in a peer-to-peer network and we use it to create a back-end for a car sharing application that will not rely on centralized structures. We will use Chord, a protocol of a peer-to-peer distributed hash table, on top of a publish-subscribe messaging system. This will allow us to have a good routing between nodes and a fast results from a query. Finally, we will experiment on the scalability of our design.

Chapter 2

Background

In this section we introduce the necessary concepts to understand the design of our protocol.

2.1 Publisher-Subscriber model

Publish-subscribe (Pub-sub) is a model of a messaging system where people, the publishers, post messages and subscribers receive messages. Publishers do not specify to whom the messages will be delivered but instead specify the topic of their messages. In the same fashion, subscribers do not a specific person but select the topics of the messages they want to receive.

2.2 Distributed hash table

A distributed hash table, or DHT for short, is a system that is distributed in different computers, or nodes, in the same network. It provides the network a fast way to locate information distributed among nodes. A DHT contains key-value pairs and every node can access the value associated with a key. Each key is unique and maps to a value which is the actual data one wants to store over the network.

We define the keyspace as the set of all possible keys. The keyspace is divided into disjoint sets and each node is responsible for one of them.

Every node has a set of neighbors that the node can send messages to. The topology of the network depends on the implementation of the DHT.

When a node sends data over the network, the following happens: something relative to the data, like a filename, will be hashed using a SHA-1 hashing function which produces a key k that

falls inside the keyset and a message *post(k, data)* is sent to the node that is responsible for the key *k*. The message is forwarded through the network using a routing algorithm proper to each DHT implementation. The responsible node then stores the key and the data. Nodes can now access the data by hashing the filename to obtain the key k and sending a message *get(k)*.

2.2.1 Chord

Chord is distributed lookup protocol for a distributed hash table. Chord states that nodes are given identifiers that fall in the same set than the keyspace, the keyspace being a m-bit number. That means that nodes' ids are distributed between 0 and $2^m - 1$, m must be large enough to make sure no pair of nodes share the same id. Nodes are sorted by their identifiers and organized as a *Chord ring*. A key *k* is then assigned to the first node whose identifier is greater or equal to the key.

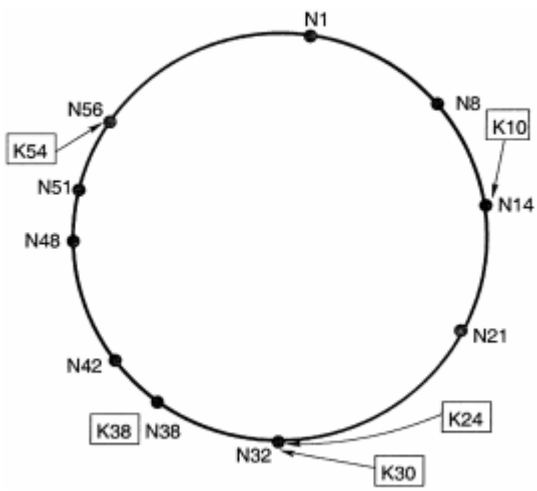

Figure 2.1: Chord ring with m = 6, 10 nodes and 5 keys. Every key is assigned to the next node clockwise from it. [3]

Each node keeps a *finger table* holding at most *m* entries. Each entry is the address of a node inside the network. The fingers are defined as the first node on the circle that succeeds $(n + 2^{k-1})$ mod 2^m, $1 \leq k \leq m$, n being the node identifier. When a node wants to get the data from a key *k*, it transfers the message to the closest node to its destination that is inside its finger table. With this method, a node can reach out to any other node in the network with only $\mathcal{O}(logN)$ hops, N being the total number of nodes.

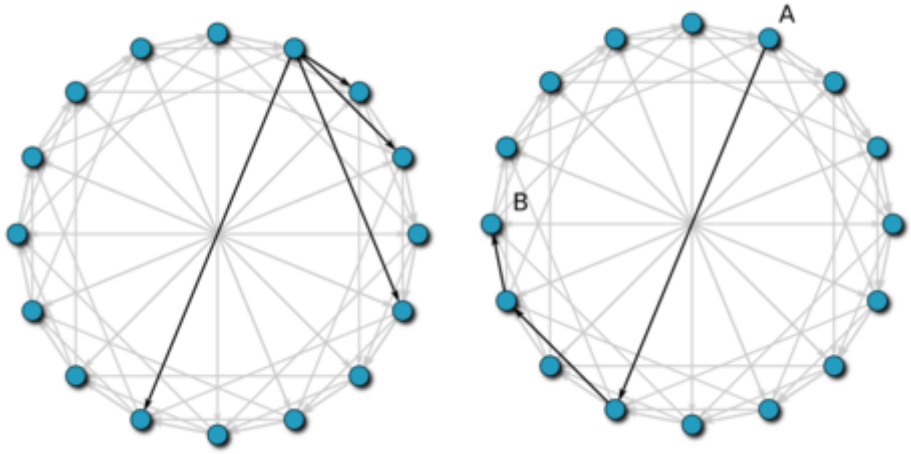

Figure 2.2 (on the left): Fully populated chord ring with m = 4. The connections of one of the nodes are in bold. Figure 2.3 (on the right): Node A sending a message to node B with the minimal number of hops needed.[1]

[1] Illustrations from `https://en.wikipedia.org/wiki/Chord_(peer-to-peer)`

Chapter 3

Design

Now let's introduce the decisions that shaped our design during this project.

3.1 Protocol choice

We considered our problem to be solvable using an implementation of pub-sub. Ride suppliers and ride seekers would take the roles of publishers and subscribers. They would publish what they search or what they offer and subscribe for any interested party. The *topic*, in terms of pub-sub, will be one of the variables of the ride, namely the hour of departure, the origin or the destination.

We chose to add Chord on top of pub-sub to have a protocol to store and retrieve information easily. By using Chord, we make sure that the information is disseminated evenly over the network and every bit of information is stored once. The key used by the DHT will be the same as the topic for pub-sub and will be generated by hashing the topic using the SHA-1 hash function. Nodes will be assigned a unique number when they are created[1], this number will also be hashed to determine their id and thus, their position inside the *Chord ring*. The keyspace will be the space containing all 32-bit numbers rendering the probabilities of collision between nodes almost zero. Having a keyspace of high cardinality also helps lowering the chances of two nodes having their ids from the hash too close from each other. If two nodes ids are too close on the *Chord ring*, the one that come before the other on the *Chord ring* will be responsible for many more keys than the one that comes after.

[1] Going from zero to the number of

3.1.1 Choice of the key for the DHT

Since the key is the result of a hashing function, the distribution of the data inside the network is already good. However if the set of inputs for the hashing contains duplicates, the distribution of information will be worsened. Let's suppose we use the hash of the departure time as the key. There would be spikes of offers and demands during certain times of the day. Now let's exaggerate the phenomenon and suppose all the rides all depart at 7 AM and at 5 PM to simulate people going and leaving work all at the same time. That would mean that only two nodes would be handling all the traffic for the day. For that reason, we selected the departure location of the ride to be input for the key as we supposed this would be the ride's element that would be the most evenly distributed.

3.2 Matching

The matching algorithm is what will notify drivers and passengers that they have found someone. A typical matchmaking process will go as the following: a passenger is looking for a ride and a driver is offering a seat.[2] The location they want to start the ride is the same. The location gets hashed and generates the same key. Their messages are routed through the network and what we will call the *matchmaking node* receives both of them (1). It then evaluates if the offer and the request are compatible by checking if the proposed and requested starting time of the course, starting place and arrival place are close enough.[3] If the two are close enough, it sends to the ride seeker a copy of the offer inside a match (2). When the ride seeker receives a match, it waits for a bit to let time for other potential incoming matches to come. When that time is up, it then selects the best match from all the offers related to its request and sends a proposal to the driver having offered the best course (3).[4] The driver's node then checks if it, in the mean time, has not found a passenger already and has no remaining seats left. If a seat is still available, it contacts its newfound passenger to seal the deal and notifies the matchmaking node that its offer is now stale and must be removed (4). All the forementioned messages are routed through Chord's protocol.

[2] A visualization of the messages sent is available in figure 3.1 below.

[3] Note that since the key for the DHT is the hash of the starting location, the matchmaking node will always receive offers and requests that have the same starting location.

[4] Since our program does not implement user interaction during this phase, it automatically selects a driver using a function we will detail in the next chapter

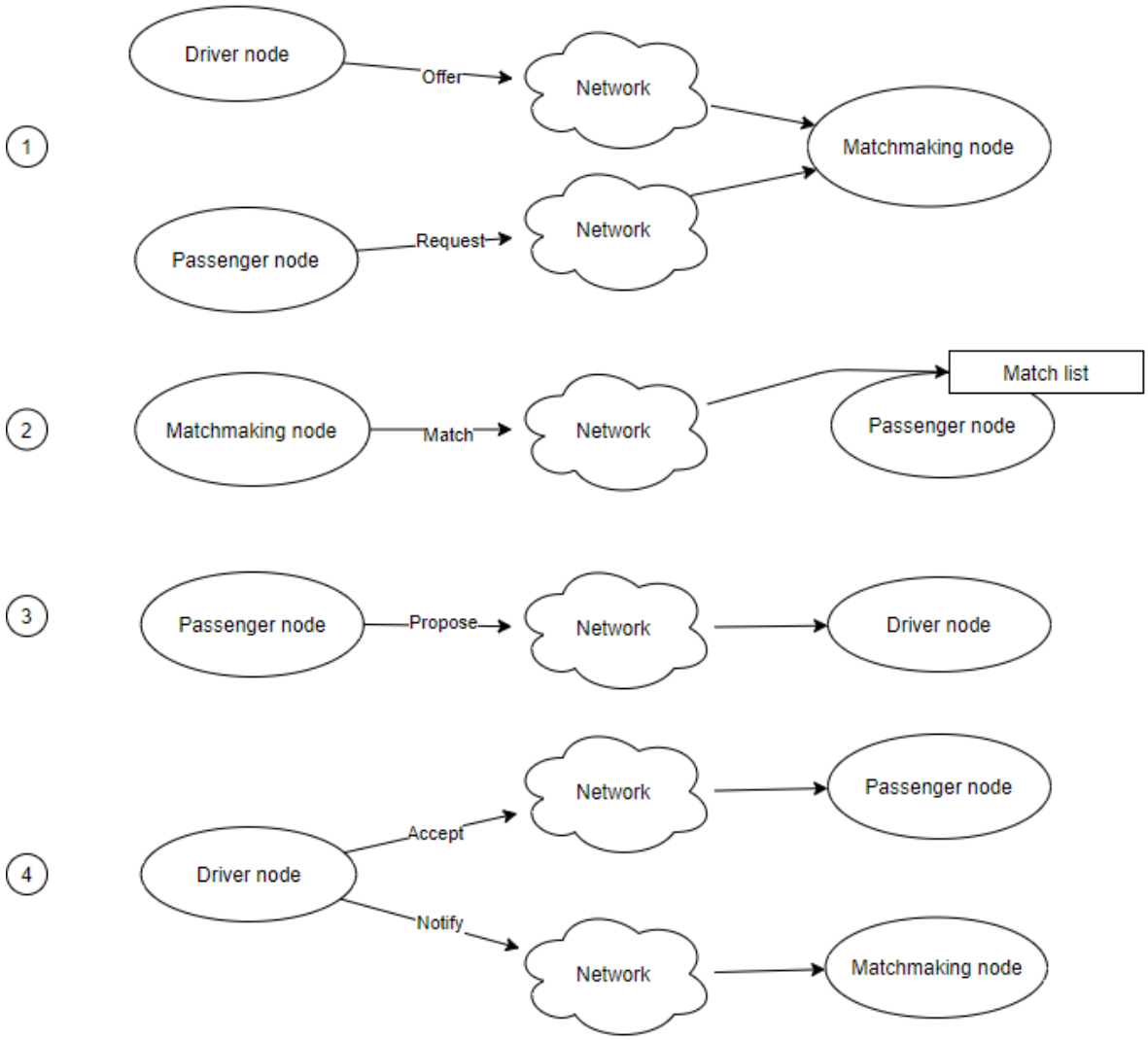

Figure 3.1: Illustration of interactions between actors

3.2.1 Location precision trade-off

Since the program generates keys for by hashing the starting location of the ride and only matches people having the same key, the question of how treat the location remains. Our idea was to considerate the world as a set of sectors by mapping the continuous GPS coordinates to discrete coordinates that would be transformed using a function that bijectively maps two values into one[5]. But how to discretize them ? If the sectors are too small, drivers and passengers would almost never obtain matches because the probability of entering the same coordinates to the meter is unlikely. On the other hand if the sectors are too big, users would receive lots

[5]Such as Cantor's pairing function

of non-revelant matches and the network would get saturated as a single request generates hundreds of matches. In our design we chose to split the world in squares of 0.01 degrees of latitude by 0.01 degrees of longitude. In our latitudes it corresponds roughly to one kilometer squared sectors.[6]

[6] Please note that edge-cases such as selecting places near the poles have not been considered

Chapter 4

Implementation

Now let us discuss how our design has been implemented

4.1 API

Our objective is to give to the user of our system a simple way to offer and find a ride. This objective is carried out by the introduction of a *(simplified)* API:

offer(departure time, departure place, arrival place) allows the user to inform potential ride seekers that a course will be available by specifying when and where he will set off and where he is headed.

request(departure time, departure place, arrival place) allows the user to search for a ride the matches his needs.

The program we made is built in Python to fully take advantage of the available libraries for the language.

4.2 Classes

Several classes have been implemented to facilitate the development of the protocol

- Request: Contains the id of the passenger, the identifier of the request[1], the departure time, departure location and arrival location

[1] A 128-bit number randomly generated

- Offer: Contains the id of the driver, the identifier of the offer, the departure time, departure location and arrival location
- Match: Contains the ids of the driver and the passenger, the identifiers of the offer and the request and also the departure time, departure location and arrival location proposed by the driver[2]

4.3 Network

Every node has 2 sockets. One is used for sending messages and the other is used to receive incoming messages. The listening socket possesses a queue of incoming connections[3]. If this queue is full and there is an other incoming connection, the socket fails and drops it.

4.3.1 Protocol

Every message is sent as a JSON encoded strings. To help nodes distinguish different types of messages, we added a *mode* field that can be the following:

- OFFER: The message from a driver containing an *offer*
- REQUEST: The message from a ride seeker containing a *request*
- MATCH: Message generated by a matchmaking node to the attention of the ride seeker containing a *match*
- PROPOSE: Message addressed to a driver from a passenger asking if a slot is still open
- ACCEPT/REFUSE: Response from the driver to the passenger following a PROPOSE message
- NOTIFY: Message from the driver to the matchmaking node informing that the offer is now invalid

4.3.2 Event loop

We use the *socket* library of python to communicate between nodes. Each node is permanently inside an infinite loop that waits for incoming connections using a blocking method. When a

[2]The ride's details requested from the passenger are not present because a match is directed to the potential passenger. All requests (and also offers) posted by a node are stored locally to retrieve requests' info to select the best match

[3]The typical length of this queue is 5 on an average machine

node receives a message, it first extracts the *mode* of the message and checks if the message is addressed to it. If that is the case, it processes the message. If not, it forwards it to the closest node to destination.

4.4 Match evaluation

Matchmaking nodes consider an offer and a request compatible using hyper-parameters, by default nodes send a match if the hour of departure do not differ from more than an hour, the departure and arrival places are within a 5-kilometer square. Those values are high on purpose to let the user have a big pool of matches.

As we said in the previous chapter, no user interface exist to let the user select the ride it prefers. Instead, passenger nodes select the best course by rating all matches they received using the following computation:

match score = (1 + L1-Norm(proposed location, requested location))[4] * (1 + time difference)[5]

[4]We used the L1-Norm because going from point A to point B is never straight, especially in cities and we wanted to take this into account when choosing the best match.

[5]Every 5-minute interval increment adds 1 to the count

Chapter 5

Evaluation

In this section, we perform experiments on our design and report our results.

5.1 Scalability of a DHT over pub-sub

This experiment evaluates the scalability of using a DHT to route messages from a pub-sub application. The dataset we used is a metadata file[1] of an instant messaging service that contains the following informations: when a person registered on the network and which id it was attributed and the identifiers of the sender and the receiver of the message. We can transform these data into inputs for a publisher-subscriber model where people joining the network subscribe to a topic associated with their id and people sending messages publish with the topic of their addressee. For this experiment, we used a simplified protocol that only sends requests, offers and matches. This means this is just the integration of Chord over an application using the publish-subscribe pattern. The following experiment was realized using 2000 publications and subscriptions.

[1] Available here: `http://opsahl.co.uk/tnet/datasets/OCnodeslinks_chars.txt`

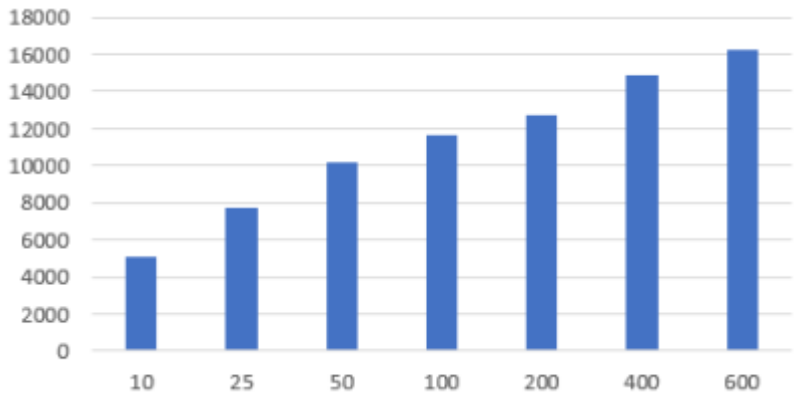

Figure 5.1: Number of messages sent in function of the number of nodes in the network.

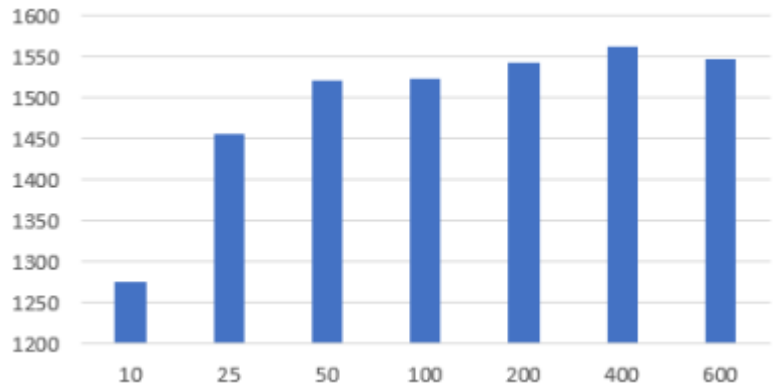

Figure 5.2: Number of unique matches sent in function of the number of nodes in the network.

Notice that the number of matches found is not the same when using different amounts of nodes. This is caused by the fact that nodes can publish and subscribe to topics without sending messages over the network. If this is the case, then the match is not counted. The probability of this phenomenon tends towards zero as the number of nodes grow. This trend is noticeable on our results.

Chord guarantees that the number of hops needed for one message is bounded by $\mathcal{O}(\log_2 n)$, n being the number of nodes. The number of unique messages sent is equal to the number of inputs plus the number of matches found. Therefore, the number of messages of our model should be bounded by $\mathcal{O}(\log_2 n * (publications + subscriptions + matches))$. When verifying if this applies in practice, we notice that the effective number of messages sent is around half the upper bound, which indeed follows our theory.

5.1.1 Stress testing a node

We performed an experiment to see if nodes were resistant to a high flow of requests and offers directed to a single node. The result is that even if 100 nodes continuously send messages to a single node without interruptions, the stressed node never fails. Although this is a good sign, we must remind ourselves that our tests were done on more powerful machines[2] than the average computer.

5.2 Scalability of our complete design

Now let's try this experiment again but this time with our full protocol described in chapter 3. The dataset we used for this experiment is the record of yellow taxi trips of New York of May 2015[3]. Here are the results for 2000 offers and requests.

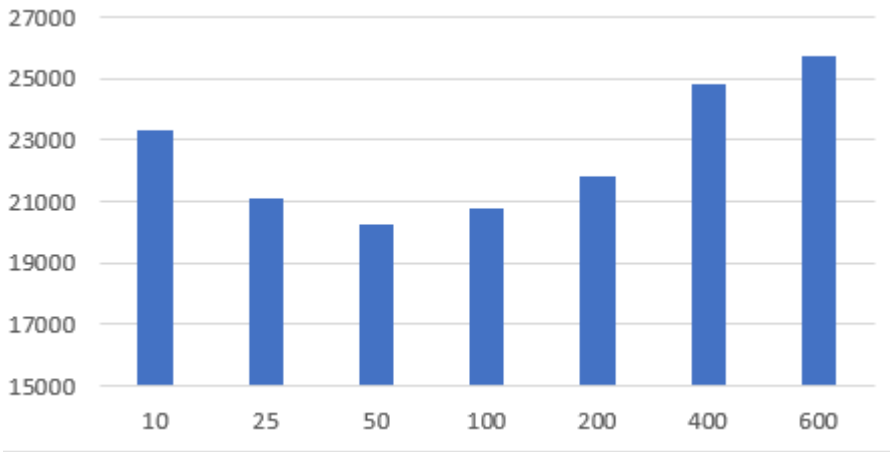

Figure 5.3: Number of messages sent in function of the number of nodes in the network.

The number of messages increase as the amount of nodes rises, but not before going down first. The increase of messages on the right side is easily explainable as messages need more hops to reach their destination. But why the amount of messages decrease at first? The upper bound of messages sent is $\mathcal{O}(\log_2 n * (offers + requests + matches + (2 * proposals^4) + rides^5))$. Let's count how many matches and how many rides are generated.

[2]The machines labostrex126 to labostrex130 provided by the EPFL.
[3]Available here: `https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page`
[4]This number is doubled because the driver responds with a positive or negative message.
[5]Which equals to the notifications sent to the matchmaker nodes.

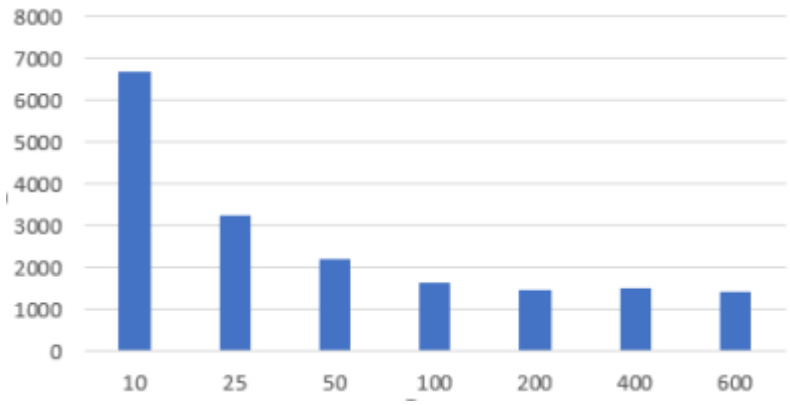

Figure 5.4: Number of unique messages sent in function of the number of nodes in the network.

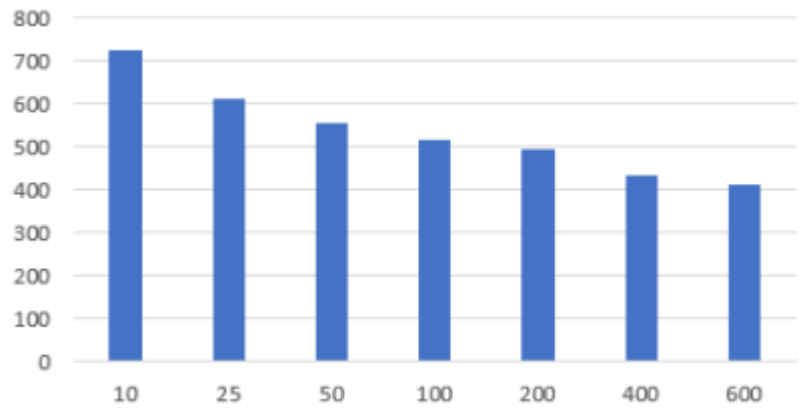

Figure 5.5: Number of rides created in function of the number of nodes in the network.

As you can see, the number of matches and rides[6] decrease when the size of the network grows. This trend did not appear during our first experiment. The difference between the two experiments is how, this time, matches are created. We previously stated that offers and requests are routed to the same node if their starting location is the same, thus creating matches. However matches can also happen when two starting locations, which are close to each other but not exactly the same, generate keys that are attributed to the same node. The chances of this happening decreases as there are more nodes in the network. We assumed that matches created from requests and offers that do not satisfy this property have less chances to be selected. This assumption was not completely wrong even though the amount of matches are divided by 5 from 10 nodes to 600 nodes, the number of rides created is only divided by 2 which means that matches are of better quality when there are more nodes.

[6]Do not confuse matches and rides. Matches are offers that are close enough of the request and rides are actual arrangements between the driver and the passenger.

Chapter 6

Related Work

Here is a list of publications that are related to our work.

6.1 MATCH: A Decentralized Middleware for Fair Matchmaking In Peer-to-Peer Markets

Like our design, MATCH [4] is a decentralized middleware for the matchmaking of ride-hailing application. MATCH do not use a DHT to disseminate requests but opts to broadcast them to a random subset of matchmakers in the network. This subset is small enough to not use too much bandwidth and large enough to have collisions where offers and requests arrive to the same node with good probabilities. MATCH also focuses on the fairness aspect of their protocol to counter malicious nodes that promote their own, sub-optimal, rides.

6.2 Scribe: The Design of a Large-Scale Event Notification Infrastructure

The publisher-subscriber pattern has existed for a long time and researchers already tackled the task to route messages efficiently using distributed hash tables. Scribe[2] is a large-scale event notification infrastructure for topic-based publish-subscribe applications built on top on Pastry[1], an other implementation of a DHT.

Chapter 7

Conclusion

We have presented a decentralized car sharing matching algorithm built on top of Chord, a peer-to-peer distributed hash table capable of routing messages through a network. Our work focused on having good matching performances and limiting the usage of messages sent over the network. With our protocol, users send offers and requests to the network and the passenger receives matches when compatible offers are available. A passenger then selects the best match and notifies the driver. We have shown experimentally the scalability of using a distributed hash table on top of pub-sub. We have also shown that our car sharing matching algorithm generates matches and routes messages on the network without creating a large amount of messages.

However, less rides were arranged when having a large number of users in the network. This is due to a non-optimal method to generate keys for the DHT. A possible improvement to this project would be to change how the keys are generated to have more matches and thus, more people sharing their cars.

In our implementation, we had a fixed network, i.e. nodes could not join or leave. This would be a great improvement to this application because real systems are not static as users can not stay connected all the time. This feature would re-distribute keys of the DHT when users leave or join the network[1].

[1] Note that only 1/n keys would be re-distributed, n being the number of nodes connected to the network. This is one of the many advantages of using a DHT

Bibliography

[1] AntonyRowstron and Peter Druschel. “Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems”. In: *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*. 2001.

[2] Antony Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel. “Scribe: The Design of a Large-Scale Event Notification Infrastructure”. In: *Networked Group Communication, Third International COST264 Workshop*. 2001.

[3] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. “Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications”. In: *IEEE/ACM transactions on networking* 11.1 (2003), pp. 17–32.

[4] Martijn de Vos, Georgy Ishmaev, and Johan Pouwelse. “MATCH: A Decentralized Middleware for Fair Matchmaking In Peer-to-Peer Markets”. In: *21st International Middleware Conference*. 2020.