Stride: A Decentralized, P2P, Rideshare System

Sujit Varadhan
University of Illinois Urbana Champaign
United States
sujitv2@illinois.edu

Pallavi Jain
University of Illinois Urbana Champaign
United States
pjain15@illinois.edu

Adarsh Suresh
University of Illinois Urbana Champaign
United States
adarshs3@illinois.edu

ABSTRACT

The dynamic rideshare problem is one in which a set of drivers and riders are constantly entering and exiting the system. Matches need to be computed such that riders get to their destination in the smallest amount of time while drivers have to travel the least amount of distance to pick up the rider. Many commercial ridesharing apps exist and attempt to solve this problem. Unfortunately, these apps are all costly to maintain, exploitative of drivers, and have large-scale privacy concerns.

In this paper, we present Stride, a P2P, decentralized ridesharing system that eliminates the presence of any main server or middleman for scheduling trips between riders and drivers. Our model provides an efficient computation of matches distributed among all peers. Since Stride is decentralized and P2P, drivers keep all income from their rides. The privacy of users is maintained through anonymization and decentralization, preventing any forms of data collection at central servers. A real-world implementation of Stride is magnitudes cheaper to host than commercial rideshare companies which spend billions of dollars each year.

We employ techniques such as distributed k-means clustering to group together geographically close riders and drivers into "bidding pools" in which an auction mechanism occurs to compute near-optimal matches in a decentralized, P2P manner.

CCS CONCEPTS

• **Distributed systems** → *peer-to-peer systems*; • **Networks** → *Location based services*; • **Applied computing** → **Transportation**.

KEYWORDS

peer-to-peer, networks, dynamic ride-sharing

CS 525 '23, Jan 17–May 03, 2023, Urbana, IL
2023. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM Reference Format:
Sujit Varadhan, Pallavi Jain, and Adarsh Suresh. 2023. Stride: A Decentralized, P2P, Rideshare System. In *Proceedings of CS 525 '23: Advanced Distributed Systems (CS 525 '23).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In this paper, we propose Stride, a system to facilitate on-demand ridesharing in a peer-to-peer, decentralized manner. Our system is novel in its real-world implementation of historical rideshare algorithms and previous theoretical work [22] [3] [16] [13]. We consider real-world implementation issues such as load balancing, leader election, failure detection, distributed location-based clustering, and anonymity.

Stride has 4 components: 1) Client Nodes, 2) Introducers, 3) Clustering Node (CN), and 4) Main Clusterer (MC). Client Nodes are either riders, drivers, or stationary nodes joining the system. Stationary nodes are users who are not requesting a ride but volunteering their device to perform computation in the system. Client Nodes can be assigned as a CN or the MC itself by spinning up a parallel process on request. This offloads computation in Stride to users of the system itself. Figure 1 summarizes the system components.

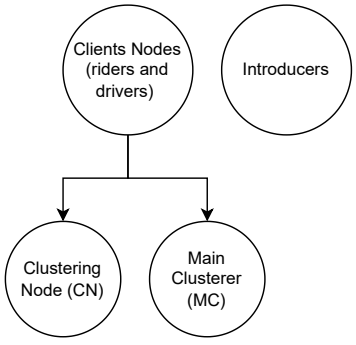


Figure 1: System Structure & Step-by-Step Process

The main job of the introducers is to simply forward client requests to one CN. We support many introducers for one Stride instance to distribute total message load. The introducers are informed of CN failures and update their membership lists accordingly.

CNs and the MC have the job of running a distributed k-means clustering protocol which clusters client nodes into similar groups based on starting location. Once clients are aware of their groups, they perform an auction/bidding mechanism (adapted from [13]) amongst each other to create close to optimal matchings between riders and drivers. These protocols combined ensure that riders and drivers who are geographically close to each other are matched.

Stride calculates a monetary cost for each ride based on the distance of each ride as well as the extra distance drivers have to travel to pick up potential rider(s). Our system does not deal with the actual mechanism of payment. We only connect potential riders and drivers who will then coordinate with each other in order to facilitate pickup and a mode of payment. A dedicated payment mechanism is omitted to reduce costs associated with running the system as well as to increase scalability.

Stride also maintains user privacy by not exposing or requiring any personal information. Users join anonymously with a universally unique identifier (UUID) generated for each session. No forms of data collection or targeted advertising are employed. Further, the decentralized nature of Stride makes it resistant to data collection inherent in other centralized ridesharing systems.

Section 2 describes the motivation behind the design of Stride. Section 3 describes other related work. Section 4 presents the overall system model of Stride in more detail. Section 5 presents the algorithms and methods used in Stride in detail. Section 6 details various performance testing and experiments used to test Stride. Section 7 discusses future work and directions for Stride. Finally, Section 8 offers the conclusion of the paper.

2 MOTIVATION

On-demand ride-sharing is a highly sought-after convenience. Apps such as Uber and Lyft are used heavily across the world and provide millions of people with a mode of transportation. The global rideshare industry is valued at $85.8 billion [1]. Additionally, 44% and 36% of the populations of China and America respectively use ridesharing [1].

Current rideshare platforms are, however, highly exploitative of drivers. On paper, Uber drivers are advertised to keep 75% from each ride [17]. However, in reality, this number is significantly lower. Rideshare apps charge many different service and booking fees that are incurred by the driver. Additionally, none of the price calculations take into account the extra cost incurred due to the distance that the driver has to travel to pick up the rider. In the end, this lowers the amount of net income that a driver makes from a single ride to around 50% to 60% of what they could have made without Uber or Lyft acting as the middleman [9] [14]. With many drivers using ridesharing apps as their primary source of income, the cuts taken by platform holders and extra costs incurred by drivers can only be characterized as exploitation.

The same booking fees that drivers have to pay are also incurred by riders. Riders are also subject to "surge pricing" which can spike the overall costs of rides exponentially [17]. These mechanisms add unnecessary monetary costs to the entire system. We believe that we can eliminate these forms of price gouging with Stride.

Drivers face problems such as not knowing the destination when accepting a rider. This forces many drivers to travel unreasonable distances away from their starting location. Drivers who do not fulfill the ride incur additional cancellation costs and may even be banned off the rideshare platform [2]. Again, with many drivers using these platforms as their primary source of income, they may feel forced into serving a ride because the consequences for not fulfilling it are severe.

We also aim to eliminate privacy concerns present in Uber and Lyft which both sell sensitive user data to advertisers to make revenue. Uber in particular projects a run rate of $500 million in ad sales for 2023 and $1 billion for 2024 [18]. We see the industry-wide increase in targeted advertising as a huge privacy concern, especially given that Uber and Lyft deal with sensitive user location data. Since our system will have no data collection, centralized servers, or profit incentive, the concern of user data being harvested is eliminated. Maintaining the anonymity of users of Stride in service of privacy is a key design consideration of our system (Described in Section 4).

Finally, we aim to eliminate the excessive monetary cost incurred by current rideshare platform hosts. While Uber has high revenue, the company has incurred net losses since 2016 totaling around $25 billion [7]. The majority of Uber's operating expenses (40%) are in credit card processing fees, data center expenses, and mobile device and service expenses [15]. Since Stride has no central servers or notion of payment processing, we

avoid these costs. The second and third biggest cost is paying employees and marketing, respectively [15]. Stride is a decentralized, peer-to-peer system with no profit incentive, thus avoiding the normal expenses that companies incur around marketing and compensation. We believe the economic excess present in current rideshare companies can be eliminated through Stride.

The end goal of Stride is to eliminate the "middleman" that today's rideshare companies operate as under the current system by making a strictly P2P, decentralized ridesharing system.

3 RELATED WORKS

3.1 The Rideshare Problem

A dynamic ride-sharing system matches and schedules rides among a group of peers somewhere in their journey. Riders and drivers are matched based on their origin and destination locations by solving a ride-matching problem. Centralized integer programming graph partitioning approaches have been proposed in [11], [8], and [21]. Amirmahdi and Neda [16] describe a trip-based graph partitioning technique where they cluster the users in R regions based on the origin and destinations of their journeys. Dividing the ride-matching problem into smaller regions analogous to sub-problems reduces the complexity of the problem. They use a "dissimilarity matrix" measure based on the shareability of trips as opposed to using a spatial measure and solve for maximizing the vehicle miles traveled savings (VMTS) periodically to keep the system up-to-date.

3.1.1 Decentralized Peer-to-Peer Shared Ride Systems. Wu, Guan, and Winter [22] ran a simulation on different ride-sharing systems in Manhattan, New York City consisting of varied ratios of private vehicles, buses, etc. In their simulation, they allow rider agents to broadcast their locations and ride requests to nearby driver agents. They test different distances to which rider agents can broadcast this information. The paper concludes that mid-range communication, i.e. broadcasting to a max of 3 radio-range hops, is sufficient for creating efficient trips without extensive flooding of the network. Additionally, this paper proposes a bidding process in which a client rider pushes their ride request to the network. Drivers in an appropriate vicinity of the rider provide offers for completing the ride. The rider then chooses the best offer for them at which point the rider and driver are connected.

Other papers about the bidding-based approach such as Nourinejad and Roorda [13] propose an "agent-based model" that combines various concepts from above. They use the rolling time horizon approach used in [4] [12] to match a pool of users who all inevitably join the system at different times within fixed time intervals. They also use a vicinity-based approach to match riders to drivers who are within the vicinity of the drivers' path. An auction/bidding phase based on a single-shot first-price Vickrey auction is also used to generate optimal results. Drivers lists are populated with potential rider matches based on the vicinity approach and ranked from best to worst based on a cost function. The cost function is determined by the proportion of the original trip distance/cost without ridesharing. Drivers will bid on riders that yield a minimum cost for the driver and riders can either accept or decline the offer they receive (based on the same cost function).

In [13], they compare a decentralized agent-based approach against the centralized integer programming graph partitioning approaches such as [16] and conclude that their agent-based approach creates near-optimal solutions while being computationally faster. Their agent-based model can also be applied to the case of multiple riders and one driver, yielding higher VMTS. As a result, we employ a modified version of the auction mechanism described in [13] in Stride.

3.2 Blockchain

In Wadi, Shidore, Surangalikar, etc., [20], they use the Ethereum blockchain as the infrastructure for a P2P system enabling ride-sharing. A rider sends a ride request which is added to the blockchain. Drivers in the vicinity of the requested ride get notified and add their bids for the ride to the blockchain. The rider chooses the bid and this agreement gets added to the blockchain. Unfortunately, the paper never describes how exactly riders in the corresponding location of a ride get notified. Additionally, the paper doesn't describe how the blockchain would scale to everyone in a country or if there would be a blockchain per geographic region such as a town, county, state, etc. The paper also does not discuss system-wide resources and computational capability needed to power their blockchain application. While a blockchain can be used to solve the rideshare problem in a decentralized manner, we believe the excess compute resources needed to power blockchains are not needed and only add unneeded complexity and cost.

3.3 Dataset

There are several datasets containing taxi and ridesharing information in various metropolitan areas. Tafreshian and Masoud [16] use the New York City (NYC) Taxi

Dataset which consists of several years of taxi operations in NYC. The paper uses rides in Manhattan during peak times. They partition Manhattan into hexagonal regions and analyze the density of pickups and dropoffs in each region. An issue with the NYC dataset is that taxi rides aren't necessarily representative of rides that occur through ridesharing services such as Uber and Lyft. The City of Chicago provides a dataset [6] similar to the NYC dataset with the difference being that it only contains trips reported by ridesharing companies. From this dataset for the month of March 2023, we found there were an average of 18,253 rides per day and an average of 787 rides per hour. We also found the busiest hour to be on the evening of March 16, 2023, with 1700 recorded rides for that hour. This gives us the average number of rides expected in a minute to be 13-30. In Section 6, we show that Stride has high performance in the average case (13-30 rides per minute) and the worst-case scenarios (busiest traffic hours in the largest cities).

4 SYSTEM MODEL AND DESIGN

The components of our system are all separated and designed to be operated in an ad-hoc decentralized manner. Our system has 4 different components: 1) Client Nodes; 2) Introducers; 3) Main Clusterer (MC); and 4) Clustering Nodes (CNs). The core execution loop of our system upon receiving a client join request is as follows (also shown in Figure 2):

(1) Clients ping a known introducer for their geographic region to join the system.
(2) Their request is forwarded to the pool of CNs.
(3) Client requests accumulate at the pool of CNs which then performs a distributed k-means clustering to group nodes with similar starting locations into the same clusters.
(4) The computed clusters are then sent by the Main Clusterer (MC) to all the clients.
(5) Clients then self-form themselves into bidding groups based on the k-means output and start an auction which results in close to optimal matchings of riders and drivers.
(6) Matched clients who are not also operating as CNs leave the system. Unmatched clients leave the system and rejoin again from Step (1).

All messages in the system are sent through asynchronous and parallel RPCs.

4.1 Clients

Clients (riders and drivers) join the system by sending a join request to one known introducer. Clients join the system anonymously to maintain the privacy of users. A UUID is randomly generated for each client that joins. Information about their current location and desired destination is sent through an RPC.

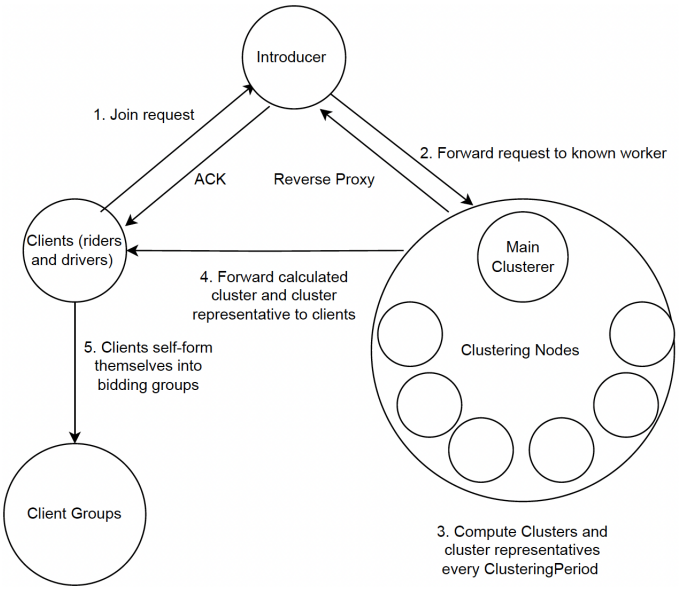


Figure 2: System Structure & Step-by-Step Process

4.2 Introducer

The introducer's job is to forward client requests to *one* Clustering Node (CN) from its pool of known CNs. This ensures an equal distribution of load amongst all CNs.

The design of the introducer ensures that it can be lightweight and requires minimal resources to host (discussed in section 6). Many introducers can be hosted for a given Stride instance. Introducer's IPs are static and assumed to be known by the clients. Clients will send a request to one introducer for their region at random. This distributes the overall load across many devices and eliminates a single point of failure.

The introducers are notified about CN failures via its reverse proxy and update its pool accordingly. The introducer also assigns incoming clients to be CNs as needed. Currently, we use a system-wide threshold for the maximum number of CNs. If this number is not met, any incoming driver nodes will be assigned as CNs. Upon being assigned as a CN, clients spin up a CN process (a parallel go-routine) and inform the introducer that they are ready to receive clustering requests.

Introducers are spun up based on region. Introducers will ignore requests from nodes outside of their specified region. For example, there could be an introducer for users in the Urbana-Champaign area and another for the Chicago metropolitan area. These separate Stride instances do not interact at all and serve completely

different geographic locations and have different computational needs and resources. This design decision is influenced by the fact that the average Uber ride is only 5.41 miles [9]. This design also ensures drivers are not forced to drive riders unreasonable distances from their starting location. Finally, this decision isolates introducers from all other Stride instances, reducing overall load and cost.

4.3 Main Clusterer and Clustering Nodes

The Main Clusterer's (MC) job is to send StartClustering requests to CNs and to take the union of the CN's partial outputs. Every *ClusteringPeriod* (set to 1 minute in our implementation, system-wide configurable parameter), the MC tells the CNs to start the process of distributed k-means clustering. Clustering is delayed by some time due to observations in previous work [22] [4]. At the end of clustering, nodes with similar start locations are clustered into groups and designated cluster representatives (centroids) are assigned per group. Nodes that are closest to the center of the cluster are designated as the cluster representatives. The output of the distributed k-means clustering is combined at the MC. Clients are then informed of which cluster they belong to and the members of their cluster. Clients then begin the auction process amongst themselves.

The MC and CNs are placed in a virtual ring to facilitate failure detection and leader election. Lightweight Ping and ACK messages are exchanged in order to validate the liveness of CNs. Node failures and node joins are propagated to the MC, other CNs, and the introducer. Upon failure of the MC, a leader election process is initiated to elect a new MC.

The task of clustering is placed on drivers in the system. We strategically choose drivers to take on the role of CNs because they are likely to stay online in the system for longer periods of time than riders (most drivers do not stop after serving one ride). Additionally, drivers are likely to be plugged into a constant source of power (car charger or portable battery). We also leave an option for nodes to join the system as stationary nodes (not a driver nor a rider) whose sole job is to perform distributed clustering operations. While there is currently no incentive for joining as a stationary node, we believe that leaving this option open is still important. Funding for stationary nodes could be acquired through crowdfunding platforms, donations, or philanthropic users of Stride who choose to use their machine as a stationary node.

4.4 Client Bidding Pools

Once the distributed clustering concludes, clients self-form themselves into "bidding pools" based on the k-means clustering output. Clients are aware of all the members in their pool. An "auction" process starts in which close to optimal pairings of drivers and riders are calculated (described in Section 5). At a high level, drivers place "bids" on riders that yield a minimal cost for them. Cost is defined by how much extra distance drivers have to travel to pick up the rider(s). Riders then can either accept or decline a bid. The bidding process is adapted from [13]. If the rider does not respond or is already matched, the driver places a bid on the next lowest-cost rider in their list. Timeouts are used to prevent drivers from waiting on riders that may have left the system. This keeps the auction process lightweight and constantly moving forward (no hanging on potential rider matches).

Once clients are matched, communication information is exchanged between the pairs and the nodes leave the system (if they are not also CNs). Clients that are not matched are given up and re-enter the system to be clustered again. This choice, while seemingly counterintuitive, simplifies the system design and computational cost significantly. Maintaining persistent bidding pools would require some notion of a global state and a consensus protocol. Having bidding pools collapse after one round of matching avoids having to incur the computational and message cost of extra consensus mechanisms. Since Stride will be running primarily on mobile devices, we want these processes to be as lightweight as possible. Additionally, the number of unmatched riders and drivers is relatively low and the distributed k-means clustering protocol remains performant at large numbers of nodes which further eliminates the need for persistent bidding pools (Section 6).

5 ALGORITHMS

5.1 Failure Detection and Membership Lists

After clustering nodes are assigned, nodes need to maintain a full membership list to perform failure detection. This needs to be achieved using minimal bandwidth along with a low failure detection time. To achieve this, we organize cluster nodes into a virtual ring of nodes. The introducer sends the ring membership list to each node in the ring when it joins. Each node then initializes its virtual ring and responds to the introducer with an ACK. The newly joined node spins up its threads to send PINGs and accept PINGs and then notifies the

introducer that it's ready to send and accept pings. The introducer then notifies all other nodes that they can start sending pings to the newly joined nodes. This sequence of steps helps prevent race conditions of other nodes sending pings to the newly joined node before the newly joined node is ready to accept pings.

Each node runs the PING-ACK process on its predecessor and successor every four seconds. The PING-ACK process consists of sending a PING message and expecting an ACK message in return. If an ACK message isn't initially received, the failure-detecting node retries at most two more times with an exponential backoff period. It first waits two seconds, then four seconds. We found that having a single retry does not suffice and frequently results in inaccurately detected failed nodes. If a node is detected as failed, the failure-detecting node tells every other node in its cluster that its neighbor failed. The PING-ACKs occur over UDP to have low bandwidth usage as these messages are sent constantly. This is at the expense of accuracy since a node might be marked as failed when its ACK just took longer to reach or its ACK got dropped through multiple retries. We account for this by having nodes remove themselves from the cluster if they receive a message saying that the node that failed is itself. Node failure messages are sent using RPCs over TCP as these messages aren't as frequent and the delivery of these messages is important to maintain consistent membership lists across nodes in the cluster.

We choose to have a complete failure detector rather than an accurate one due to the following rationale. In terms of the end system, an incomplete failure detector could potentially send a clustering request to a failed clustering node which leaves a subset of riders and drivers in an unmatchable state. This significantly reduces the throughput of the system in terms of driver-rider matches. On the other hand, a failure detector that doesn't have 100% accuracy could mistakenly identify a clustering node as failed. This will increase the overall load on the other clustering nodes as there are less working clustering nodes in the system. This might increase latency of the system, but riders and drivers will still be eventually clustered and matched. We decided that we would rather have eventual matches than no matches. Thus, we implemented a complete failure detector instead of one that focuses on accuracy.

We considered implementing failure detection for the resultant clusters/bidding pools but decided against it. The rationale was that bidding pools are intentionally ephemeral because we want bids to be agreed upon and matches to occur as fast as possible. Therefore, we wouldn't gain much from failure detection, especially at the expense of increased bandwidth. If an undetected failure of a driver or rider does occur, they can rejoin the system in the next round of clustering. As commercial ride-sharing matching occurs on the order of several minutes, this isn't much of an overhead for end-users.

5.2 Distributed K-Means Clustering

Clusters represent groups of clients with similar start locations. We decided to do the clustering based on start location coordinates alone. The rationale behind this is based on [22] which finds that matching nodes with close start locations is sufficient for generating optimal matches. After considering several theoretical and practical clustering techniques, we decided on implementing k-means clustering.

We adopt the distributed coreset construction algorithm outlined in [5] to implement distributed k-means clustering. In this approach, a *coreset* is a weighted set of nodes that has a cost on any set of centers that is approximately equal to the cost of the original nodes. Using distributed clustering, sharing all the data can be avoided by creating a smaller summary of the critical information. Each site can summarize its data and send it to a central coordinator. This strategy enables us to solve the problem using less data, which reduces the amount of communication required. As described in Section 4, we have several clustering nodes and one Main Clusterer (MC) node. The MC node indicates the dedicated clustering nodes (CNs) to start the clustering process every *ClusteringPeriod* time period (set to 1 minute in our implementation).

The distributed k-means clustering technique is a 2-step process that involves running the coreset generation algorithm on the given CNs to generate local coresets and then combining the results of the CNs together to form global coresets and clusters at the MCs.

The Distributed coreset construction process [Algorithm 1] is split into two phases. In the first phase, we use Lloyd's algorithm to calculate k centers, determine the cost of approximating the centers for the given membership list of client nodes, and then use a message-passing algorithm (implemented as an RPC) to communicate this information to all other CNs. In the second phase, we use the communicated cost to generate local coresets at each CN. First, we calculate the total number of client nodes to be clustered in this iteration, denoted by t, by combining the length of membership list information received from all other CNs currently performing k-means clustering. Then, using the weighted cost of client node points at each CN, we determine the number of client nodes to be clustered by each CN, denoted by t_i. We randomly sample

non-uniformly from the local client node data points to select t_i nodes from the membership list to create a coreset based on their approximated costs/weights. We combine the k-centroids from the first step with this coreset to create the final local coreset at each CN. Each CN returns its local coreset to the MC.

Algorithm 1 Distributed coreset construction

Input Parameters: Local membership list of client nodes (start location coordinates), k where k=number of clusters
Output: Coreset
Algorithm:

- Phase 1:
(1) Perform k-center clustering on node data points using Lloyd's Algorithm
(2) Compute the cost of finding the centers for the given points
(3) Communicate the cost and length of the membership list to all other CNs.
- Phase 2:
(1) Compute the total number of nodes to be clustered by this CN
(2) Set weights on local data
(3) Random sampling of t_i points to get a coreset
(4) Include local centroids from the first step of phase 1 to the coreset.

Algorithm 2 Distributed clustering on coreset

Input Parameters: Local membership list of client nodes (start location coordinates), k where k=number of clusters
Output: Global clusters
Algorithm:

- Phase 1: on each CN, construct its local coreset using Algorithm 1
- Phase 2: on Main Clusterer node,
(1) Union of local coresets to get Global coreset
(2) Lloyd's clustering algorithm applied on the global coreset to compute the centers
(3) Other nodes attached to the final clusters using clustering result from Step 1 of Algorithm 1.

In our implementation, the k-means clustering algorithm considers whether incoming client nodes are drivers or riders and assigns them weights accordingly. The algorithm clusters the nodes while aiming to achieve a balanced distribution of drivers and riders within each cluster, according to a predetermined ratio called the $d2r - ratio$. This approach ensures that clusters do not consist solely of drivers or riders, which could result in no match being made between them. To maintain this ratio, we have implemented a weighted version of the k-means algorithm that aims to cluster the nodes into k clusters. The $d2r - ratio$ is treated as a constant in our implementation to simplify the process.

The process begins by constructing local weighted coresets at each CN. These coresets are then used by the MC to create a global coreset [Algorithm 2]. This global coreset replaces the original data and can be used with any distributed clustering algorithm. The MC combines the coresets from each node and applies a weighted k-means clustering algorithm to obtain the final clusters. Client nodes that are not included in the global coreset are assigned to clusters based on the results of Step 1 of the distributed coreset construction. The MC sends the final clustering result and assigned the cluster back to the client nodes.

To understand the process, consider an example where we have 3 clustering nodes (CNs), 20 client nodes, and the number of clusters, $k = 3$. The introducer randomly assigns these 20 client nodes to the 3 CNs. Let's assume that the three CNs receive 6, 6, and 8 clients respectively. These clients get added to the membership lists of the respective CNs. The MC initiates the k-means-clustering on each of these CNs and waits for them to return their local coresets. Let's take the third CN who got 8 clients to cluster. Following the Algorithm 1, first, we get 3 out 8 clients as the cluster centroids with their corresponding clusters. We compute the cost (distance of the clients from the nearest centroid) of the 8 client nodes and the total cost. This cost is communicated to the other 2 CNs. Similarly, the other 2 CNs also find their centroids and calculate and communicate their cost to this all the other CNs. Next, we calculate t and respective t_i's for each of the CNs based on the cost of all their nodes. Let's take the derived t_i to be 1,1,2, respectively for our three CNs. We then randomly sample the t_i nodes according to their cost and they represent the initial coresets. Then we merge the k-centroids, i.e., the 3 centroid nodes to the initial coreset. As a result, we get a coreset with $k + t_1 = 3 + 1 = 4$ nodes for CN1, a coreset with $k + t_2 = 3 + 1 = 4$ nodes for CN2 and a coreset with $k + t_3 = 3 + 2 = 5$ nodes for CN3. These local coresets are returned to the MC and unioned into a global coreset of $4 + 4 + 5 = 13$ nodes and applied a weighted k-means to cluster them into $k = 3$ clusters. The remaining 7 nodes which are not part of the global coreset are assigned to the cluster according to the clustering result of the first step when each CN calculated their 3 centroids and their corresponding clusters. For example, on CN3 with 8

client nodes $n_1, n_2 ... n_8$, local k-means-clustering yielded 3 clusters $c_1 : [n_1, n_2, n_8], c_2 : [n_3, n_4, n_7], c_3 : [n_5, n_6]$ represented by centroids $cd_1 = n_1, cd_2 = n_4, cd_3 = n_5$, respectively. Its local coreset included 5 nodes (3 centroids and 2 derived from random sampling of t_3 points), let's say, $n_1 ... n_5$. The remaining non-coreset nodes that do not participate in the final k-means-clustering on Main Clusterer are $n_6 .. n_8$. Node n_7 that belonged to cluster c_2 (represented by centroid cd_2) in their local k-means-clustering will, in final k-means-clustering on Main Clusterer, be attached to the cluster to which the centroid node n_4 (because $cd_2 = n_4$) is clustered into.

5.3 Auction and Bidding

Once clients are assigned to bidding pools, they perform an auction mechanism amongst each other where drivers place bids on riders that yield them the lowest cost. If a rider declines a bid, the driver moves on to the next lowest-cost rider in their list. Figure 3 describes the flow of the auction mechanism.

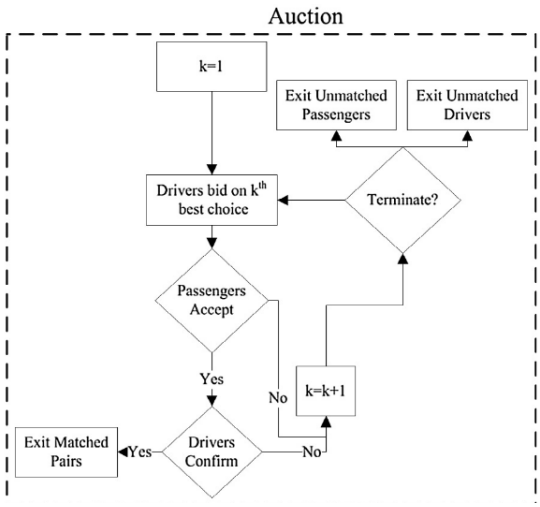


Figure 3: Auction process between drivers & riders

In Stride, there are separate cost functions used by the driver and rider. The cost borne by the driver is the extra distance that they have to drive from their current location to pick up the rider. Drivers select riders based on which of them yields the least cost for them. The cost incurred by riders is the driver's cost plus the distance from the rider's start to their desired destination. Riders can either accept or decline a bid based on this cost. Currently, trip pricing is calculated by taking the associated rider cost and converting it to a dollar amount at a \$1 = 1KM rate. This means that riders pay for their portion of the trip as well as the extra distance that drivers have to drive to pick them up. This differs from traditional ridesharing apps, which do not consider the additional costs borne by drivers.

6 EVALUATION

6.1 K-Means Clustering

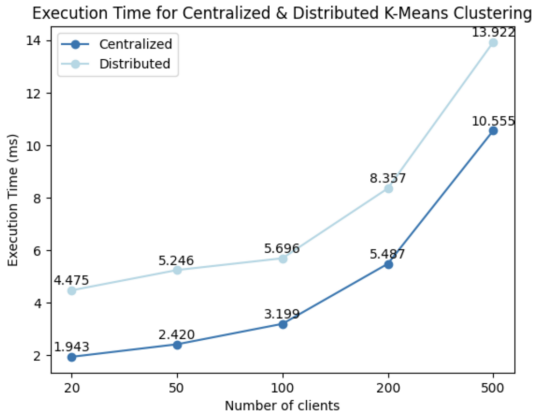


Figure 4: Execution time for centralized v/s distributed k-means-clustering

For conducting our k-means clustering experiments, we assign virtual machines (VMs) to our computing nodes (Introducer, Main Clusterer(MC), and Clustering Nodes(CNs)) and the client nodes (Riders and Drivers) in the system. Each of our VMs is a 2CPU core @2.1GHz each, 4GB RAM machine, all present at a single virtual farm. We ran two simulations to evaluate our k-means-clustering algorithm on the Chicago Taxi Trips dataset provided by the City of Chicago [6]. We took the trip data from the month of March 2023 from the Chicago dataset and compare the performance of a centralized k-means-clustering approach with our distributed k-means clustering algorithm. Next, we simulate the busiest hour of Chicago from March 2023 in terms of the number of trips completed to evaluate the performance of our clustering algorithm.

We implemented distributed k-means-clustering and compared our results with the centralized k-means-clustering for the same system setup as mentioned above. For this, we took the starting location coordinates of random trips from the March 2023 trips dataset and ran clustering algorithms to cluster the clients into 5 clusters with a maximum of 10 CN instances. Figure 4 shows that the distributed k-means took slightly more time (order of 3-5 milliseconds) than the centralized approach. Distributed clustering incurs extra computation costs as it performs message passing among the CNs and an extra round of k-means in MC. But the performance is not significantly worse. The main benefit of this is to avoid a single point of failure which is worth the tradeoffs in the context of our system which expects users to wait around 1 minute to be matched anyways.

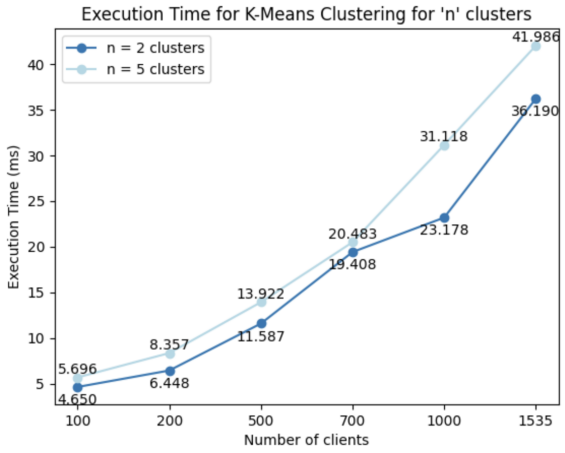


Figure 5: Execution time of k-means-clustering for clustering client nodes into n clusters

The next simulation involves mirroring the busiest hour in March 2023 in Chicago. As mentioned under Dataset (related works section 3.3) before, we use these findings as our base. Our distributed k-means-clustering algorithm runs every minute, and if it can handle the hourly load within a minute with decent performance, it would demonstrate the potential of our algorithm. We evaluated the execution time (in milliseconds) of our algorithm for different loads, ranging from 100 to 1535 client nodes clustered into 2 or 5 clusters. Figure 5 shows the performance of our algorithm. Stride is capable of clustering around 1500 client nodes in 35-45 ms, where the load is 60 times the expected load from the Chicago dataset [6] seen in a minute. The gain in execution time from clustering into 2 to 5 clusters is not significant because the algorithm steps are mainly invariant to communication cost, and the computation costs only depend on distance calculations for each cluster. The VM specifications we used are comparable to or slightly worse than modern smartphones. Despite this, our algorithm's execution time is impressive. All VMs used in our simulation were running on the same farm, resulting in low message-passing latencies. In a real-world scenario where mobile devices are not in the same location, we expect the execution time to increase slightly. However, our algorithm's performance for the busiest hour concluding within a minute can compensate for the increased execution time in mobile devices.

6.2 Failure Detection

We ran an experiment to calculate the average convergence time of a cluster after n simultaneous failures of adjacent nodes in the virtual ring. The system converges when all the live nodes in the system receive updates of all the failed nodes and update their membership lists. When adjacent nodes fail, this acts as a worst-case scenario since it takes longer to detect n adjacent failures than n failures spread out through the ring. For example, if three adjacent nodes fail, the middle node has no nodes detecting its failure as its two adjacent nodes also fail. As a result, we have to wait for the ring to update with the failure of the two nodes at either end of the three adjacent failed ones in order for the remaining live nodes to run failure detection on the middle node and realize that it failed. On the other hand, if the three failed nodes were spread out through the ring, all three nodes could have their failure detected at the same time because all three nodes have live nodes adjacent to them that are detecting their failures.

Additionally, the worst-case behavior of n adjacent nodes failing has a low probability. To explain this, the probability of n nodes failing simultaneously is inversely proportional to n. Moreover, if n arbitrary nodes fail simultaneously in a ring of N nodes, the probability that those nodes are all adjacent to each other is $\frac{N}{\binom{N}{n}}$ as there are N ways such that the n failed nodes are adjacent in the ring and $\binom{N}{n}$ ways that n nodes could fail in the ring.

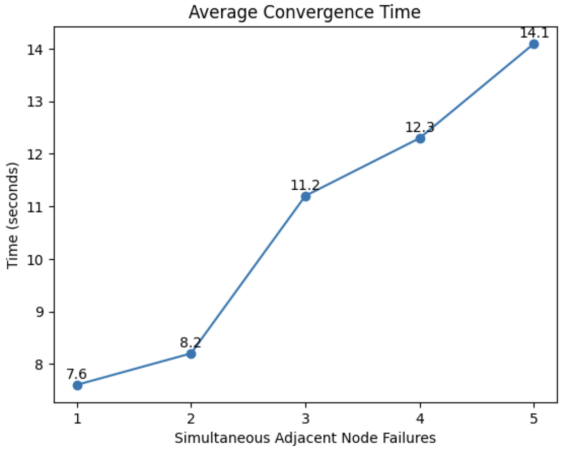
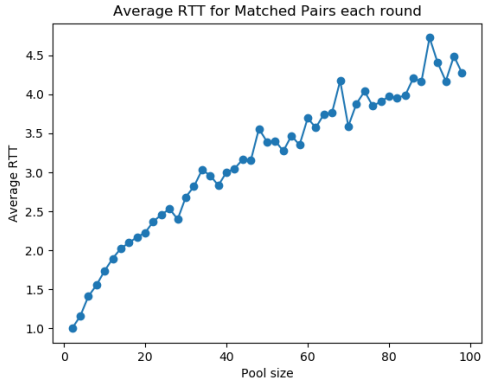


Figure 6: Average Virtual Ring Convergence Time after Failure

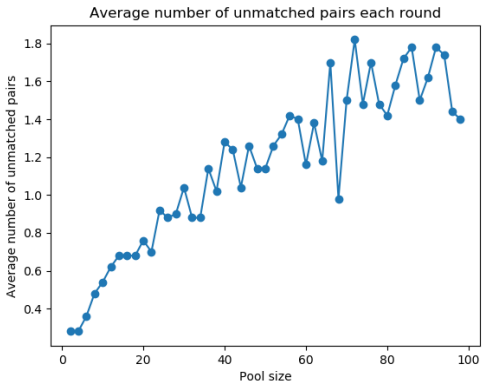
The convergence time for two adjacent nodes is around eight seconds. This is due to the retries and exponential backoff periods. In terms of P2P systems, this time is relatively high. However, with regard to user experience in the final system, eight seconds is a fraction of the time it usually takes to request a ride from a ride-sharing app. Since the alternative is to have more frequent PING-ACKs, fewer retries, and shorter backoff periods which increases bandwidth and the likelihood of false positives which immediately decreases user experience, this higher failure detection time is justifiable.

6.3 Auction and Bidding

In order to test our auction mechanism, we ran two simulations on different real-world scenarios as well as a system test with real machines.



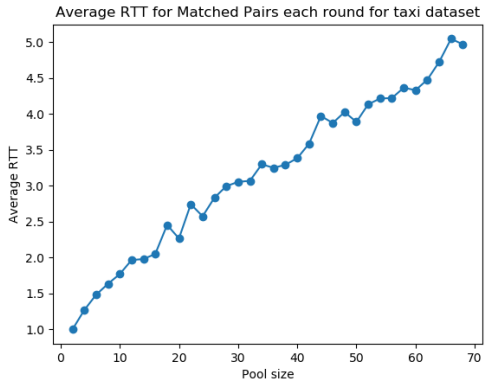
(a) Average RTTs Needed to Complete Matchings Between Riders and Drivers



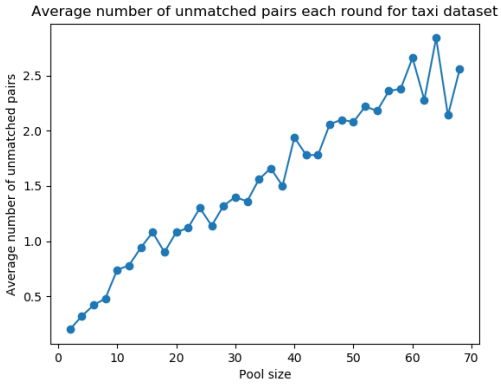
(b) Average Number of Unmatched Pairs in a Bidding Round

Figure 7: Simulation Results using Randomly Sampled Points in the Champaign-Urbana Area

Our simulation mirrors a real Stride bidding pool by creating a group of drivers that place bids on the same pool of riders. We simulate each pool with an equal number of riders and drivers. While this may not be a realistic scenario, we use it to purely measure the performance of our auction mechanism since we cannot take into account real-world driver supply and we do not consider carpooling for our current implementation (discussed in section 7). We take into account inherent network delays present in mobile networks by adding a randomized delay (mean: 50ms, standard deviation: 25ms) for each bid that is sent. The maximum cost that a rider is willing to pay was simulated by taking the distance they want to travel and adding a randomized distance from 5KM to 40KM. This accurately maps the cost we impose on riders in Stride as the distance they want to travel plus some extra distance that they make the driver travel to pick them up. We used a pricing scale of 1KM = $1.



(a) Average RTTs Needed to Complete Matchings Between Riders and Drivers



(b) Average Number of Unmatched Pairs in a Bidding Round

Figure 8: Simulation Results Using the Chicago Taxi Dataset

The first simulation sampled random points in the Champaign-Urbana area and assigned these as the driver/rider start and destination points. Figure 7a shows the average number of round trip times (RTT) that were taken to match a driver with a rider. 1 RTT in our system would be the time from a driver sending the bid and receiving the response. The RTTs scale sub-linearly after bidding pools of size 20. This provides strong evidence for the protocol's scalability. Even in the largest cases (n=100), the message latency of around 4 RTTs is a relatively short amount of time in the real world. Figure 7b demonstrates the relatively low amount of unmatched pairs in each bidding round. Even in large bidding pools (n=100), the number of unmatched pairs on average is never larger than 2. This further supports Stride's design decision to not maintain persistent bidding pools and require unmatched nodes to rejoin the system. The extra load incurred by this design decision is minimal.

We also run this same simulation but with the Chicago Taxi Dataset [6]. This provides a relatively good measure of rideshare demand. It also provides a good worst-case scenario for our system since Chicago is one of the busiest cities in the U.S. We also strategically chose the busiest time of day with the most rides served from the dataset to better model a worst-case scenario for our system. Figure 8a shows the average RTTs needed to match a rider and a driver. This simulation shows a more linear relationship than the previous simulation. However, even in the largest bidding pools (n=70), only 5 RTTs on average are required. Figure 8b also shows a slightly worse performance than the previous random simulation. However, in the largest bidding pools (n=70), an average of less than 3 unmatched pairs still will not impose any significant extra load on our system.

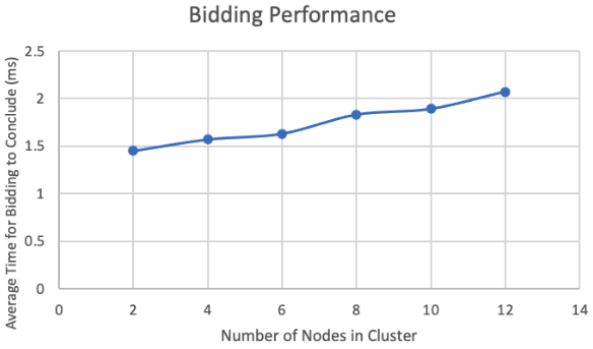


Figure 9: Average Time to Find Matches on Real Hardware

We also ran tests on real hardware to see what real-world performance looks like (Figure 9). We ran our tests on AlmaLinux virtual machines with a dual-core Intel(R) Xeon(R) Silver 4216 2.10 GHz CPU, 4 GB RAM, and 60 GB storage. These technical specifications are comparable to what's found on modern smartphones which provides confidence in the computational feasibility of our system. Although we were only able to test on bidding pools with a maximum size of 12, we still see a similar trend of the auction mechanism taking little time to complete with performance on the order of milliseconds. In the future, we hope to expand testing to mobile devices that are geographically far away from each other to gauge what the real-world latency of Stride will be.

6.4 Monetary Cost

We ran a simulation of our auction mechanism on the Chicago Taxi Dataset and measured the average cost of rides in our system using the $1 = 1KM metric. Figure 10 shows an interesting trend of constantly decreasing trip costs as the bidding pool size increases.

The main cost of hosting our system emerges from the introducer(s) per region. The introducer doesn't perform much computational work but performs a lot of network I/O. If we take an 8 vCPU and 32 GiB EC2 virtual machine on AWS and use it for a year, the cost is $6009.36 according to AWS' price estimation calculator. Uber spent 8.9 billion dollars on core platform infrastructure in 2021 [15]. According to Uber's website [19], they operate in over 10,000 cities which results in an operational cost of $890,000 per city. We can host multiple introducers per city and still operate our system at a lesser price. Since we offload our computation to drivers, we don't have to host and manage our own VMs. This significantly reduces the cost of running our system and lets all of the revenue go to the drivers.

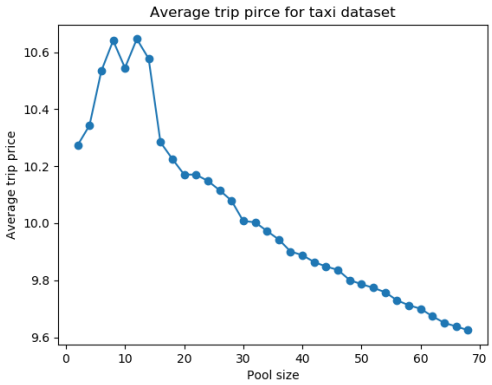


Figure 10: Average Price per Ride

7 DISCUSSION & FUTURE WORK

7.1 Safety of Users

One limitation of our system is the potential safety concerns that arise due to privacy-preserving mechanisms in Stride. Every user joins the system with a randomized UUID. This means that there is no good way of verifying the identity of anyone on the system which could lead to safety concerns.

One way to ensure safety in Stride could be to have users submit themselves for a one-time background check. The non-personally identifying results of the background check could then be digitally signed using public/private key encryption and the introducers could verify the authenticity of background checks. These digital signatures could also act as a form of security to prevent malicious actors from overwhelming the system with false requests. One drawback of this is the one-time cost incurred by users of Stride by purchasing a background check. Another drawback is the increased workload that the introducers take on by verifying a user's digital signature.

7.2 Decentralized Introducers

Another limitation of our system design is the presence of points of centralization in the introducers. While the introducers are designed to be lightweight and have a relatively low cost to host (see Section 6), it goes against our key principle of decentralization.

Currently, we distribute the load of the introducer by having many known introducers for a single geographic region. All of the introducer IPs are static and are assumed to be known by the clients. Clients then send a join request to one of the known introducers at random. Introducers run a failure detection protocol along with the CNs and MC where the introducers are informed of CN failures.

In the future, we aim to make the introducers purely decentralized. We would allow any person to contribute their machine as an introducer node rather than just hosting the introducers on a cloud platform. A new introducer (I_{new}) would join by sending a request to another known introducer in the network (I_{known}). I_{known} then adds INEW to the failure detection ring and informs all the other introducers about I_{new} existence. Clients send requests to one introducer that they know about. The introducer then sends back a full membership list of all introduces in the system and the client updates its list of known introducers accordingly. One issue in this approach is the fact that when all introducers go offline the entire system goes offline too. Another issue is that there is no direct incentive for a user to contribute their machine as an introducer.

7.3 Other Planned Features

We intend to improve Stride's cost function by incorporating location-accurate trip time estimation using the Open Source Routing Machine (OSRM) [10]. The solution by [10], while computationally intensive, is very fast and does not require the Internet. Currently, cost is only a function of distance traveled. By utilizing trip time estimates, the costs we generate are a better indication of the real cost incurred by the driver. Every minute that a driver has to wait in traffic is another minute that they could serve a different ride. We intend to incorporate trip time estimation as a supplementary determiner of cost in our bidding protocol. We also plan to experiment with the cost function by adding metrics such as a minimum ride cost and price adjustments based on driver supply and rider demand.

In the future, we intend to extend Stride's bidding protocol to support carpooling (many riders per driver). This extension is discussed in [13] and allows a driver to place a bid on pairs of riders instead of just a single rider. One challenge to this approach would be creating meaningful incentives for a driver to pick up multiple riders and modifying our cost function to reflect the increased distance driven.

We also intend to expand our system to accommodate food delivery. The food delivery industry has similar problems to ridesharing for both drivers and riders. The food delivery problem also maps neatly to the rideshare problem. Food orders can simply be treated as riders with the start point being the restaurant and the end point being the customer's food drop-off location.

7.4 The Fallacy of Incentives

We believe that incentives are not always required in systems and in society in general. Many facets of our society such as open source software and volunteer work run without concrete profit incentives. Additionally, many industries and professions such as education, academia, musicianship, and artwork have limited compensation and run mainly on passion.

While some aspects of Stride do not have any incentives behind them (hosting an introducer and participating as a stationary node), we believe that some users may still choose to contribute their computing resources or money solely because they believe in the vision of Stride. As a result, we have designed our system such that users can "volunteer" their computation. Stride, however, is not reliant on these volunteers because the majority of computation is performed by users of Stride itself. Stride is also designed to be lightweight and scalable.

8 CONCLUSION

We present Stride, a decentralized, peer-to-peer system to facilitate ridesharing. Our system is highly distributed. The job of clustering can be spread across an arbitrary number of nodes. Users of Stride itself take on the computational workload and contribute directly to the upkeep of the system. Additionally, the protocols we use are lightweight. The auction mechanism is fast to compute close to optimal matchings and is computationally inexpensive. Our implementation is also highly scalable. The core protocol can be run to serve different regions and sub-regions. Instances of Stride are isolated and can be spun up anywhere as long as there are users using the system. Most importantly, Stride solves key issues that plague commercial rideshare apps such as privacy concerns, exploitation of drivers, and high cost of maintenance. We hope that the design of Stride can inspire the shift away from centralized corporate control and the rise of new peer-to-peer systems designed for the people, not profit.

REFERENCES

[1] 2023. 20+ riveting ridesharing industry statistics [2023]: Average ridesharing revenue, market share and more. https://www.zippia.com/advice/ridesharing-industry-statistics/#:~:text=The%20global%20ridesharing%20market%20is,who%20used%20them%20in%202015.

[2] News4 Anchor amp; Transportation Reporter Adam Tuss. 2016. Uber drivers stiff passengers after finding out final destination. https://www.nbcwashington.com/news/local/uber-drivers-stiff-passengers-after-finding-out-final-destination-2/127513/

[3] Niels Agatz, Alan Erera, Martin Savelsbergh, and Xing Wang. 2012. Optimization for dynamic ride-sharing: A review. *European Journal of Operational Research* 223, 2 (Dec. 2012), 295–303. https://doi.org/10.1016/j.ejor.2012.05.028

[4] Niels A.H. Agatz, Alan L. Erera, Martin W.P. Savelsbergh, and Xing Wang. 2011. Dynamic ride-sharing: A simulation study in metro Atlanta. *Transportation Research Part B: Methodological* 45, 9 (Nov. 2011), 1450–1464. https://doi.org/10.1016/j.trb.2011.05.017

[5] Maria-Florina Balcan, Steven Ehrlich, and Yingyu Liang. 2013. Distributed k-means and k-median clustering on general communication topologies. In *NIPS*.

[6] City of Chicago. 2023. Transportation network providers - trips: City of Chicago: Data Portal. https://data.cityofchicago.org/Transportation/Taxi-Trips/wrvz-psew

[7] /entity/gennaro Cuofano. 2023. Is uber profitable? uber profitability 2016-2022. https://fourweekmba.com/is-uber-profitable/

[8] Charles M Fiduccia and Robert M Mattheyses. 1982. A linear-time heuristic for improving network partitions. 19th—DAC—. In *IEEE Design Automation Conference.-1982.*

[9] Harry@theridesshareguy.com. 2021. Just how far is your uber driver willing to take you? https://www.forbes.com/sites/harrycampbell/2015/03/24/just-how-far-is-your-uber-driver-willing-to-take-you/?sh=6207f80d597c

[10] Stephan Huber and Christoph Rust. 2016. Calculate Travel Time and Distance with Openstreetmap Data Using the Open Source Routing Machine (OSRM). *The Stata Journal: Promoting communications on statistics and Stata* 16, 2 (June 2016), 416–423. https://doi.org/10.1177/1536867X1601600209

[11] Brian W. Kernighan and Shou-De Lin. 1970. An efficient heuristic procedure for partitioning graphs. *Bell Syst. Tech. J.* 49 (1970), 291–307.

[12] Alexander Kleiner, Bernhard Nebel, and Vittorio Amos Ziparo. 2011. A Mechanism for Dynamic Ride Sharing Based on Parallel Auctions. (2011).

[13] Mehdi Nourinejad and Matthew J. Roorda. 2016. Agent based model for dynamic ridesharing. *Transportation Research Part C: Emerging Technologies* 64 (March 2016), 117–132. https://doi.org/10.1016/j.trc.2015.07.016

[14] Quora. 2018. What percentage cut does uber take from the total fare cost of a ride? do they subtract a flat fee for each dispatch or a percentage? https://www.quora.com/What-percentage-cut-does-Uber-take-from-the-total-fare-cost-of-a-ride-Do-they-subtract-a-flat-fee-for-each-dispatch-or-a-percentage-Are-there-initiation-monthly-fees-to-be-a-driver

[15] Adhik Ramanathan. 2023. What are the key drivers of Uber's expenses? when will the company break-even? https://www.trefis.com/no-login-required/Lnu6SlSX

[16] Amirmahdi Tafreshian and Neda Masoud. 2020. Trip-based graph partitioning in dynamic ridesharing. *Transportation Research Part C-emerging Technologies* 114 (2020), 532–553.

[17] Uber Technologies. 2023. https://www.uber.com/us/en/marketplace/pricing/

[18] Uber Technologies. 2023. Uber and lyft claim big gains for their ad businesses. https://www.wsj.com/articles/uber-and-lyft-claim-big-gains-for-their-ad-businesses-2b5e0ef7

[19] Uber. 2023. Use Uber in cities around the world. https://www.uber.com/global/en/cities/

[20] Sarvesh Wadi, Mrunal Shidore, Malhar Surangalikar, Devarshi Talewar, and Vedant Savalajkar. 2022. P2P Ride-Sharing using Blockchain Technology. *INTERNATIONAL JOURNAL OF ENGINEERING RESEARCH TECHNOLOGY* 11 (sep 2022). https://doi.org/10.17577/IJERTV11IS090045

[21] Chunjiang Wu, Shijie Zhou, Linna Wei, Jiaqing Luo, Yanlin Wang, and Xiaoqiang Yang. 2007. A New k-Graph Partition Algorithm for Distributed P2P Simulation Systems. In *International Conference on Algorithms and Architectures for Parallel Processing*.

[22] Yun Wu, Lin Guan, and Stephan Winter. 2006. Peer-to-Peer Shared Ride Systems, Vol. 4540. 252–270. https://doi.org/10.1007/978-3-540-79996-2_14