SHARING ECONOMY-BASED ON-DEMAND PEER-TO-PEER TUTORING
AND RESOURCE SHARING

BY

CHINMAY SANJAY KULKARNI

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2016

Urbana, Illinois

Adviser:

Associate Professor Hari Sundaram
ABSTRACT

The sharing economy is a socio-economic ecosystem built around the sharing of human and physical resources. This is considered to be a new and alternate socio-economic system which is currently in its early stages and has tremendous potential. The sharing economy combines the need to direct demand towards supply by adding the point of view of collaboration based on the social networks and preferences of various entities involved. This is based on the economic model of Collaborative Consumption in which participants share access to resources rather than having individual ownership over them.

We propose the novel idea of a sharing economy-based model for knowledge sharing amongst peers in a classroom setting. Students often find themselves stuck on trivial implementation details like syntax, best practices and which tools to use. Many times, help is difficult to find, even though the solution might be known to one of the student’s peers. Moreover, many large classes do not have enough teaching assistants to help out students and the use of such a platform can be useful to offload simple questions within the classroom itself, saving office hours for more complex questions. This, coupled with the benefits of a collaborative learning environment for students, has motivated us towards the development of an on-demand peer-to-peer tutoring and knowledge sharing platform.

Such a platform can also help us understand how different incentive mechanisms motivate people to share knowledge: Are people motivated by social reputation or money? We develop an Android application called “Quet” which can be used by students to request help from their peers for questions related to their coursework on-demand. Preliminary observations show that an application like Quet is useful;
and we wish to deploy this in multiple courses in subsequent semesters to realize its full potential and utility to students and instructors alike.
To my parents, for their love and support.
ACKNOWLEDGMENTS

I would like to thank my adviser Associate Professor Hari Sundaram for his invaluable support and advice during my Masters degree. He helped ensure that I had all the resources and guidance I needed to complete my project. I would like to thank Pranay Prakash, who helped me develop the frontend for the mobile app Quet, which I created as a prototype for my thesis work. I would like to acknowledge the support provided by Professor Kevin Chang and John Lee in reaching out to students. Finally, I would like to thank my family and friends for their love, support and encouragement.
# TABLE OF CONTENTS

**LIST OF TABLES** ................................................................. viii  
**LIST OF FIGURES** ................................................................. ix  

**CHAPTER 1 INTRODUCTION** ................................................... 1  
1.1 Motivation ........................................................................ 2  
1.2 Brief Overview of Sharing Economy-Based Products .......... 6  
1.3 Contributions .................................................................... 11  
1.4 Thesis Outline ................................................................... 13  

**CHAPTER 2 RELATED WORK** .................................................. 15  
2.1 Resource Sharing in Social Networks ................................. 15  
2.2 Incentive Models ............................................................... 17  
2.3 Stable Marriage Problem .................................................... 23  

**CHAPTER 3 QUET: THE ON-DEMAND PEER-TO-PEER KNOWLEDGE SHARING APP**  
3.1 Overview of Quet ............................................................... 24  
3.2 Sample Working of Quet .................................................... 30  
3.3 Incentive Model ................................................................. 48  
3.4 Matching Algorithm .......................................................... 51  
3.5 System Architecture and Design ........................................ 70  

**CHAPTER 4 EXPERIMENTAL SETUP** ....................................... 81  
4.1 Survey Questions .............................................................. 81  
4.2 Performance Metrics ........................................................ 83  

**CHAPTER 5 CONCLUSION** ..................................................... 92  
5.1 Future Work ...................................................................... 93
LIST OF TABLES

1.1 An Overview of Startups using the Sharing Economy Model. . . . . 7
LIST OF FIGURES

3.1 Different possible options for the mentor matching criteria. 27
3.2 Different possible options for the mentee matching criteria. 28
3.3 Help-request session payment calculation. 29
3.4 Quet landing page. 31
3.5 Statement of consent page. 32
3.6 Error received if the user does not consent to use Quet. 32
3.7 A sample confirmation email sent once the user consents to use Quet. 33
3.8 The login page. 34
3.9 Editing your profile. 35
3.10 User’s profile page/home screen. 36
3.11 Making a help-request. 37
3.12 The match notification and screen change once Chinmay’s help-request has been accepted by Bill. 39
3.13 Chinmay and Bill use the in-app chat to decide when and where to meet up. 40
3.14 Chinmay sees the rating page to rate his mentor Bill. 41
3.15 The match notification and screen change once Chinmay’s help-request has been matched up to Bill. 43
3.16 Bill accepts Chinmay’s help-request and is thus able to see Chinmay’s information and chat with him. 44
3.17 Bill’s screen as he chats with Chinmay using Quet’s in-app chat. 45
3.18 Bill’s screen after Chinmay has pressed “Start Session”. Note that the “Cancel” button is replaced with the “End Session” button. 46
3.19 Bill sees the rating page to rate Chinmay. 47
3.20 Flowchart describing the help-request session between Bill and Chinmay. 47
3.21 Example of mentors lying on the Pareto frontier and those that are dominated for a given help-request object. 56
3.22 Example of mentees lying on the Pareto frontier and those that are dominated for a given mentor.  
3.23 Preference lists obtained for different mentee matching criteria for candidates shown in figure 3.21. 
3.24 Preference lists obtained for different mentor matching criteria for candidates shown in figure 3.22. 
3.25 Example of the social network obtained by modeling help-request sessions between users in Neo4j. 
3.26 Example of a Helped relationship wherein user 2 has helped user 1. 
3.27 Quet’s system architecture diagram. 

4.1 Profile load times without taking into account network latency. 
4.2 Profile load times taking into account network latency. 
4.3 Simple profile edit times without taking into account network latency. 
4.4 Simple profile edit times taking into account network latency. 
4.5 Heavy profile edit times without taking into account network latency. 
4.6 Heavy profile edit times taking into account network latency. 
4.7 Time taken to run a typical match query in Neo4j to find nodes and a particular “Helped” session between these nodes. 
4.8 Time taken to create (if not present) nodes for the mentor and mentee and then create a “Helped” relationship containing information mentioned in subsection 4.2.2, amongst these nodes.
CHAPTER 1
INTRODUCTION

Nowadays, there are many applications of the sharing economy model for providing and distributing services among consumers. Most of these applications are examples of Commercial Peer-to-Peer Mutualization Systems (CPMS) which provide a web-based platform to match demand and supply. Users on these applications generally provide services on a peer-to-peer basis, which means that the same set of users are service providers in some cases and consumers in other cases. Uber [44], Airbnb [1] and TaskRabbit [42] are examples of popularly used services based on the sharing economy model. Section 1.2 describes and compares various sharing economy startups. Though most sharing economy-based products are based on users exchanging services and resources that are physically quantifiable, the same model can be extended to knowledge sharing.

In large classrooms, students are often unable to get help from instructors due to insufficient office hours. Moreover, due to a large number of students, office hours are often consumed by simple and repetitive questions which students can easily solve amongst themselves in a peer-learning environment. A lot of times, students find themselves stuck on trivial implementation details, which may not even be directly related to the concepts taught in the course. Finding professional tutors is often too expensive for students and does not make sense for trivial issues. Time wasted searching the web can be better utilized in learning more complicated aspects of the course. In such a classroom setting, students can benefit from knowledge sharing and peer-to-peer studying to a great extent. Getting on-demand help and learning from peers can be very useful in such a scenario. Not only does such a service allow students to receive help from their peers on-demand, but it also helps promote
collaboration amongst peers and provides students opportunities to learn by teaching.

One of the critical challenges in designing such an application for on-demand tutoring and knowledge sharing, is promoting participation. Chapter 2 goes over the research in the field of designing incentive mechanisms on such resource sharing networks. A lot of work has been done in promoting participation on peer-to-peer content delivery and online information sharing networks. Most of this work has dealt with using social reputation or money to incentivize participation. Another interesting problem is how to match supply and demand in the context of a sharing economy service. Finding matches that are beneficial for both the producer and consumer of the service is of utmost importance.

In this thesis we have developed an Android application called “Quet” for on-demand peer-to-peer tutoring to solve the above-mentioned problems. In order to tackle the challenges mentioned previously, we have aimed at incorporating an incentive mechanism within Quet, which motivates students to participate as much as possible, while increasing their personal profit. Another interesting problem that we have aimed at addressing, is how students requesting help on Quet are matched with the best available peer. Quet takes into account both peer’s preferences and finds the best available match so as to satisfy these preferences as much as possible.

1.1 Motivation

In this section, we present the motivation behind our research on a sharing economy-based on-demand peer-to-peer tutoring and resource sharing application. We present the research questions that we have attempted to tackle and the motivation behind the design of Quet.
1.1.1 Research Questions

Our research is aimed at answering 2 main questions:

1. How do different incentive models motivate people to share knowledge in a peer-to-peer on-demand setting?

2. How useful is an application like Quet in solving problems faced by students in a classroom setting? Can using an application like Quet motivate students to collaborate more with their peers? Can it improve their performance in the classroom?

By applying our study of on-demand peer-to-peer resource sharing in the context of tutoring and knowledge sharing for a classroom setting, we were able to try to address a multitude of problems faced by students and instructors. Finding a solution to these problems was the main motivation for developing Quet.

1.1.2 Problems Addressed through Quet

Some of the major problems we have aimed to address while designing Quet are as follows:

- **Poor student-instructor ratio**: Traditional classrooms are often faced with the problem of an imbalance in the number of enrolled students to the number of available teaching assistants and instructors. This in turn leads to dissatisfied students unable to get help when required, and overworked course staff. Quet attempts to solve this problem by allowing students to request help on-demand, without having to wait for office hours, thus reducing the pressure on course staff and also improving the student’s learning experience.

- **Ineffective office hours**: To add to the above-mentioned problem of already insufficient office hours, office hours are also often taken up by problems that can be easily solved by students in peer-to-peer learning groups. Simple questions related to implementation details, such as which resources and tools to use,
syntax-related questions, etc. often take up office hours without actually offering much help to students. Quet aims at solving this problem by encouraging students to ask questions like these, via the app to their peers. By offloading simple questions amongst peers in the classroom, we hope to see an improvement in the quality of office hours by allowing the course staff to focus on more important and difficult questions.

- **Online help often isn’t on-demand:** Online forums and websites like Piazza [32] are often not on-demand given the number of questions posted by students on these platforms. This is again due to a poor student-instructor ratio. Moreover, many posts are repetitive and instructors waste time redirecting questions towards earlier posts. Quet attempts to solve this problem by matching help-requesters up with peers on-demand.

- **Difficulty expressing questions:** Students often find it difficult to express their questions in a written form on online forums like Piazza. Moreover, a lot of students also avoid asking questions in class due to fear of embarrassment. It is often easier for students to express such questions in-person using an application like Quet. This also allows for follow-up discussion on these problems.

- **Difficulty finding help:** Even if students decide to ask for help amongst their peers, it is often difficult to find a peer that can offer help. Knowledge is hidden amongst a student’s peers in a classroom, so it becomes increasingly important (and difficult) to connect a student with an apt peer who is capable of helping the student. Quet simplifies this process by providing students with the ability to simply request help, without worrying about finding a mentor on their own.

- **Tutors are expensive:** Professional tutors are expensive. The main usage of Quet is aimed to be towards simple problems that students find themselves struggling with in their coursework. These problems are generally very specific and not necessarily conceptual questions that require the expertise of a tutor or instructor. Given the nature of these problems, it does not make sense to spend a large amount on tutors. Quet allows users to ask for help from their
peers at a minimal cost (equivalent to about the cost of a coffee). After all, a student receiving help from his or her peers would offer them a cup of coffee anyhow. The added advantage of this model is that tutoring becomes more informal and friendly.

- **Time is wasted searching the web**: Students often waste time searching the web for problems they encounter in their coursework. Many times, they may not even be able to express their problem in the form of a search query. This not only wastes time, but can be frustrating. Even if a student is able to express his or her search query correctly, it is likely that they will not receive the best possible answer catered to their needs by searching on the web. If the student was to ask his or her peer instead, it is more likely that the peer understands their question better given the context of their common courses. Not only does this save the student’s time wasted on web searching, but personal interaction also often makes problem solving easier. Quet aims at leveraging these benefits of personal interaction and contextual similarity between peers to match students up with those peers who would be able to solve their questions to the best of their abilities.

- **No sense of community within peers**: Individual study often leads to a lack of a sense of community amongst peers. Many works of research such as Boud et al. [5] and Topping [43] emphasize the importance of peer learning and how it has shown to improve student performance to a great extent. Quet implicitly aims at inculcating and strengthening this sense of community amongst peers as well.

- **Lack of exposure to teaching**: Students often do not get the opportunity to teach. By introducing an application like Quet, students are able to benefit from this opportunity. Furthermore, by allowing students to mentor their peers, it forces them to solidify their understanding of the course material. Quet hopes to improve student performance through this philosophy of “learning by teaching”.

5
• Lack of incentives to help peers: In the current classroom setting, there is barely any motivation for students to help their peers. Quet introduces incentives in the form of both money as well as quantifiable social reputation. Students willing to offer help altruistically, are allowed to do so and are in fact more likely to be matched with help-requesters. Quet aims to motivate students to share knowledge by helping their peers via this incentive mechanism.

1.2 Brief Overview of Sharing Economy-Based Products

In order to understand how resource-sharing should be tackled while designing Quet, it is important to understand similar sharing economy-based applications. In this section we present a brief overview of existing products and startups that are based on a collaborative consumption, or sharing economy model. We analyze the kinds of resources that these different startups share, as well as the nature of the resource sharing. This overview will help us understand the extent to which sharing economy models can help solve daily problems and also how they differ in the incentive mechanisms used.
<table>
<thead>
<tr>
<th>Company Name</th>
<th>Resource Shared</th>
<th>On-Demand/Planned</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uber/Lyft/Ola</td>
<td>Time, Car</td>
<td>On-Demand</td>
<td>Car owner gives rides to users on-demand. The car owner may or may not be spending time (depending on whether the person to pick up wants to go along their original route or not), but the car owner uses his or her own car.</td>
</tr>
<tr>
<td>Airbnb</td>
<td>Time, Rooms</td>
<td>Planned</td>
<td>House owners lease parts of their house or their entire house to the consumer. The host spends some time on cleaning their house, providing facilities to their tenants, etc.</td>
</tr>
<tr>
<td>JustPark</td>
<td>Space</td>
<td>Planned, On-demand</td>
<td>Users rent out their parking spaces to others based on predefined demands.</td>
</tr>
</tbody>
</table>

Table 1.1: An Overview of Startups using the Sharing Economy Model.
Instacart | Time, Car | Planned | A grocery-delivery service where customers get their groceries delivered to them. Now Instacart even has independent contractors who pick up groceries and deliver them to customers.

Vinted | Clothing | On-demand | Allows users to sell or swap secondhand clothing. If swapped, the company gets no profit. If sold, it gets some percentage.

Spinlister | Bicycles, skiing equipment, etc. | Planned | Allows users to rent bicycles or skiing equipment from other users.

LeftoverSwap | Food | Planned | An application to share leftover food with your neighbors.

RelayRides | Car | Planned | Connects users who want to rent cars inexpensively, to users who are willing to rent out their cars.

Dogvacay | Time | Planned | Allows users to find people willing to take care of their dogs while they are away on vacation.

<p>| Table 1.1 (cont.) |</p>
<table>
<thead>
<tr>
<th>Service</th>
<th>Type</th>
<th>Availability</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Streetbank</td>
<td>Things, Time</td>
<td>On-demand</td>
<td>Allows users to rent things from their neighbors who are willing to share. Neighbors can share ‘skills’ as well i.e. provide help for tasks like do-it-yourself things, gardening, etc. The application process is indicative of the fact that the focus is more on sharing things and not time.</td>
</tr>
<tr>
<td>TaskRabbit</td>
<td>Time</td>
<td>On-demand, Planned</td>
<td>Users post tasks that they want to get done (can be anything from getting coffee delivered, to groceries, to filing tax returns). Volunteers choose whichever tasks they are willing to do and share their time to do the user’s tasks. Users can even ask for virtual task completion (help coding up their websites, help in research projects, etc.)</td>
</tr>
<tr>
<td>Etsy</td>
<td>Things, Time</td>
<td>On-demand, Planned</td>
<td>Allows users to buy and sell handmade or vintage items.</td>
</tr>
</tbody>
</table>

Table 1.1 (cont.)
<table>
<thead>
<tr>
<th>Skillshare</th>
<th>Time</th>
<th>On-demand</th>
<th>Allows users to learn about varied topics via video lectures made by contributors who go through a training program related to the skill they claim to have.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vayable</td>
<td>Time</td>
<td>On-demand</td>
<td>Users offer their time to act as tourist guides/local experts for customers who are on vacation in their areas.</td>
</tr>
<tr>
<td>LendingClub</td>
<td>Money</td>
<td>On-demand</td>
<td>Allows users to lend money to others.</td>
</tr>
<tr>
<td>Fon</td>
<td>Home Wi-Fi Network</td>
<td>On-demand</td>
<td>Allow other users to use your home Wi-Fi network in exchange for getting to use any of Fon’s 8 million worldwide hotspots.</td>
</tr>
</tbody>
</table>

Table 1.1 (cont.)

VentureBeat [45] talks about sharing economy startups that belong to the Unicorn startup club. This article also evaluates startups based on the kinds of resources that are being shared. Accordingly, the author divides sharing economy startups as sharing resources in the following areas: Learning, Municipal, Money, Goods, Health, Space/Housing, Food, Utilities, Transport, Services, Logistics and Corporate.

We focus our attention to a sharing economy model wherein the resource that is collaboratively consumed is time and knowledge, in exchange for either money or social reputation. As part of this thesis, we develop Quet, which is an application for providing on-demand tutoring and help in a peer-to-peer classroom context. This sharing economy-based collaborative consumption application makes use of time as
the shared resource. Peer mentors are expected to also provide their skills to teach their matched mentee. The nature of the service provided requires it to be planned, but the service should be relatively on-demand, though not as much as in Uber, since there is a matching phase between prospective mentors and the mentees/help-requesters.

1.3 Contributions

We describe the contributions of this study as follows:

- We propose a novel application of the sharing economy-based model in the context of knowledge sharing. This kind of collaborative consumption service for knowledge sharing is especially interesting when applied in the context of education. A large part of this study focuses on solving problems faced by students in finding necessary help with their coursework in a novel manner.

- We make use of a combination of incentive mechanisms which rewards users for their participation not only through hard incentives like money, but also through soft incentives like social reputation. This combination of incentive models is useful for motivating different kinds of users. Making use of such a combination of incentives is useful in attracting users motivated by altruistic contributions, as well as those motivated solely by earning money.

- We propose an interesting approach to match users requesting help to those capable of providing help. Gale and Shapley [18] devised the Gale-Shapley algorithm for solving the Stable Marriage Problem. The Gale-Shapley algorithm is used for finding stable matches between two equally sized sets of elements, wherein each element provides a preference list of elements from the other set. A variant of the Gale-Shapley Stable Marriage Problem is used for matching students requesting help to available peers. This variant takes into account unequal sets, incomplete element preference lists and infeasible candidates for
each element. Preference lists are generated for students based on various factors which take into account the ratings and preferences of matched mentor peers as well as matched mentee peers.

- We create Quet - an Android application for on-demand peer-to-peer tutoring which makes use of the proposed incentive models, matching algorithm and the idea of knowledge sharing in a classroom setting. Quet is developed with a robust infrastructure and backend capable of sustaining a large number of users. It has been designed keeping in mind the scalability of the problem and user-base; and encompasses all of the necessary features for users to be able to request help on-demand to their peers and also provide help on-demand to matched peers. Although Quet has been designed keeping in mind the scope of a classroom (and the necessity for an on-demand tutoring service), it has the potential to be extended to various domains where knowledge sharing (and resource sharing in general) within social networks is the focus.

In order to design an effective and useful application, we had kept in mind several possible problems that we could have encountered when designing Quet. The analysis of these problem areas is highly correlated with the analysis of different sharing economy-based startups provided in section 1.2.

1.3.1 Pitfalls Avoided while Designing Quet

There were many pitfalls which we had to avoid while designing Quet in order to successfully tackle the problems we wished to address. Some of these pitfalls have been described below.

- *Authorizing users:* Quet has been designed to be deployed at courses within UIUC. The app ensures that only users who provide their consent to use the app are allowed to log in. Authorization emails and other authentication checks are made to ensure that users are enrolled students at UIUC.
• **Ensuring quality help from matched mentors:** Due to the informal nature of the matches and the low costs of providing help, it becomes important to match students with peers who provide quality help. Quet has been designed so that students and mentors rate each other after a help-request session. These ratings, though anonymous to the users, are used internally by Quet’s matching algorithm in order to reduce the probability of users being matched to low-rated peers. Thus, users being matched with unsatisfactory peers can give them a low rating, which would prevent those peers from being matched up as frequently in the future.

• **Matching demand and supply:** Since students will presumably use this application during their crunch times (before exams, etc.), the matching of mentee to mentor has to be done rapidly and very accurately so that a student does not end up wasting his or her time. To tackle this, Quet matches students with available mentors relatively on-demand, so that students do not have to wait too long to get matched. The matching also takes into account the preferences of the help-requester and the matched mentor to make the best possible matches.

• **Infrastructure scalability issues:** The infrastructure for this application would have to be robust to service multiple people at once. We go over the architecture of Quet in section 3.5. Quet has been designed to be scalable and to serve multiple simultaneous help-requests asynchronously. This allows for low latency in application usage as we have described in chapter 4.

### 1.4 Thesis Outline

The rest of the thesis is organized as follows: In chapter 2, we describe the related work. In chapter 3, we describe Quet in terms of its basic working, incentive model, the matching algorithm, as well as its underlying architecture. In this chapter, we also provide a demo working of the application. Chapter 4 talks about the experimental
setup and performance metrics related to the usage of Quet. Finally, we conclude the thesis in chapter 5, in which we also describe how Quet is different from existing products that offer similar services in the realm of on-demand tutoring.
In this chapter we present a survey of research on resource sharing in social networks, different incentive models and the Stable Marriage Problem. The research presented in this thesis draws its implementation and ideas from a combination of these three topics.

2.1 Resource Sharing in Social Networks

Yang and Chen [47] explore the applications of a social network-based collaboration to help people find relevant content and knowledgeable collaborators. Their work deals with the dissemination of knowledge in a social network and the effect and extent to which people are able to share this knowledge and information. The authors conclude that using a social network-based system to support interactive collaboration between peers in a virtual community can further the extent to which knowledge can be disseminated amongst peers. Though they do not address the problems of resource sharing in general, the idea of modeling interactions via social network analysis is interesting.

Bardhi and Eckhardt [4] talk about emerging trends among people to prefer using resources on a temporary lease-basis, rather than owning the resource. The authors provide a case study for Zipcar [48] for a specific car sharing access based consumption model. They study this market where no transfer of ownership takes place and only time consumption and a minimal service charge are resources that are used. The authors find that the current trend of access-based consumption is based on
collaboration between users augmented by social networks and common interests. The authors note that there has been a shift in the sociocultural politics of resource consumption and the concept of a sharing economy is becoming more prevalent. The authors also talk about how this trend is characteristic of short-term resource sharing as compared to long-term car leasing/renting. This leads to higher profit margins for the car owner over a period of time and also keeps their sense of ownership towards the car intact. The current trust-based resource sharing companies sacrifice user anonymity and privacy to some extent since potential customers can get information about their host through connected social networking sites and past reviews. The car sharing example addressed by the authors deals with consumers using cars owned by a company (such as Zipcar) rather than carpooling startups, but the basic economic analysis provided can be applied to any sharing economy-based product. In conclusion, the authors formalize the concepts of access-based consumption in terms of ownership, time, and other economic factors and contrast this with traditional resource leasing and buying.

Chard et al. [8] deal with leveraging the concepts of trust within members of a close social network to enable them to share heterogeneous resources amongst each other. The authors claim that social network links formed between people are based on physical interactions within these people, thus entailing a sense of trust amongst them. Resource sharing between these users will thus require less external motivation and persuasion. This paper also discusses different incentive mechanisms to motivate participation in such a social cloud. The authors evaluate extrinsically motivating factors like: gaining skills, increase in social reputation and traditional financial motivation. They also discuss intrinsic motivation such as a feeling of accomplishment or pride on performing altruistic tasks. In conclusion, the authors find that such a social cloud for resource sharing, based on real-life relationships between peers is useful.
2.2 Incentive Models

In order to decide the incentive model to be used for promoting participation for peer-to-peer knowledge sharing on Quet, it was necessary to understand current implementations of incentive models in various domains. By doing so, we were able to understand how incentive models are used to leverage participation and how they are used to reach an optimum state in a peer-to-peer network. Studying different incentive models also helped us understand the spread of certain behaviors in social networks due to these varied incentive models.

Hummel et al. [22] carry out a study to increase participation in a learning network (LN) by providing incentives to participating students. The authors design an incentive mechanism based on Social Exchange Theory i.e. the concept that a participant’s willingness to contribute/participate in an LN depends on his or her satisfaction with using the LN. Social Exchange Theory proposes that the relationships we choose to create and maintain are the ones that maximize our rewards and minimize our costs. According to this theory, there are broadly 4 incentive mechanisms to motivate community members to contribute:

1. Personal Access: The participant has a pre-existing expectation that he or she will receive actionable and useful (extra) information in return. This must obviously be information that is not accessible without the incentive.

2. Personal Reputation: Improvement in the reputation, visibility or status of a participant as a result of increased contributions.

3. Social Altruism: Participants understand the role of the underlying LN as a “social good” and are thus happy to participate and contribute without receiving tangible benefits.

4. Tangible Rewards: Participants get tangible assets in the form of money, articles, etc. as a reward for their increased participation.

The authors describe how introducing tangible rewards as incentives can destroy the notion of the LN; proposing that incentive mechanisms for knowledge-sharing should
match the spirit of what has to be achieved. The authors implement their incentive model in the context of an online information sharing forum, thus in their case the access to more information is more beneficial than other tangible rewards. They design an incentive mechanism by which participants are granted access to additional resources and information on their platform. In conclusion, the authors observe increased active as well as passive (lurkers) participation. It was also observed that participation is heavy-tailed i.e. only a small portion of the population has a large amount of participation, which is expected.

Cheng and Vassileva [9] talk about designing an adaptive incentive mechanism that takes into account the required demand in the network at the current time as well as the reputation of users, to decide the incentives that should be offered. In the model proposed in their research, incentives to participate in online knowledge sharing are not provided uniformly. Larger incentives are provided in initial stages until the community reaches its critical mass. Incentives are subsequently adapted so that low-quality information and services are not provided by users. This is done to avoid information overload in the social network. This adaptive incentive model was seen to successfully improve user retention and contributions as well as avoid information overload. In addition to this adaptive incentive mechanism, users are given reputation points based on the quality and timeliness of the services they provide as well as the quality of their ratings for other content. Users were assigned membership levels like gold, silver and bronze based on their reputations and level of contributions, which served as a mark of high social reputation on the platform.

An important result of this research is that micro payments involve a high cognitive cost for users (the decision of whether to carry out the transaction or of ‘consuming’ a shared resource when one needs to pay for it) and that social rewards, such as fame or status in the community can be a stronger motivator for participation since users don’t need to think too much before asking for a service or contributing. A major shortcoming of this research was the implementation of the adaptive incentive model. The number of points earned by a user for contributing are reduced if the supply exceeds demand i.e. if there are many others willing to contribute. This seems to be unfair and provides scope for improvement. Further work in this domain saw the
introduction of a game theoretic approach to designing the incentive mechanism so as to reach a mutually beneficial outcome for all parties involved (either the platform and the users, or the users belonging to either the consumer or producer sets).

An interesting application of incentive models to promote participation is crowdsourcing. Yang et al. [46] discuss an incentive mechanism for increasing the participation of users in a mobile phone sensing crowdsourcing activity. They introduce some interesting game theoretic approaches to designing the incentive mechanism. The incentive model proposed by the authors is composed of two parts: a Platform-Centric Model and a User-Centric Model. The Platform-Centric Incentive Model is based on a Stackelberg Game in which the platform is the leader and the users are the followers. The exact incentive is constant for each user in this model. Users can decide whether or not to participate based on this constant reward and the total cost they incur while participating in the activity. The authors design the platform-centric incentive model in such a way that the utility of the platform is maximized (maximum participation from users in terms of man-hours given their cost constraints). Besides maximizing the platform utility, this model also ensures that none of the users can improve their utility by deviating from the platform’s current strategy. The User-Centric Incentive Model allows each user to provide a base amount which he or she expects from the platform. The platform must provide at least this amount to the user. The model is auction-based and the incentive can be computed efficiently (in polynomial time). The notion of having separate models: one which is constant and depends only on the platform, and another that depends on each individual user’s expectation is interesting and can definitely be applied in various on-demand resource-sharing applications.

Another interesting domain for which designing incentive mechanisms is important is that of distributed and peer-to-peer resource sharing. Feldman et al. [14] model the incentive mechanism for resource sharing on a peer-to-peer network as a Generalized Prisoner’s Dilemma problem. The authors describe another interesting game theoretic approach to designing the incentive mechanism. They make use of personal history (analogous to a user’s past experiences with his or her peers in their previous meetings), shared history (analogous to the overall rating of previous inter-
actions with peers) as well as social reputation to model their incentive structure. The authors conclude that a mixed approach (using social reputation as well as hard incentives) is profitable for both increasing the level of participation on the platform, as well as for increasing the benefit to each individual user.

Ranganathan et al. [39] provide a very interesting and useful comparison of incentive mechanisms that they simulated on a distributed file sharing network. The authors model the resource sharing problem as a *Multi-Person Prisoner’s Dilemma* problem stating that people tend not to share resources in P2P networks to prevent individual bandwidth costs, though the global optimum would be reached if every user shared his or her resources. The specific scenario increased the need for a robust incentive model since users must be incentivized to share their resources at the risk of having to incur personal costs. The authors thus propose 3 different incentive schemes for achieving a global optimum, out of which 1 is a hard incentive scheme and 2 are soft incentive schemes. The different proposed incentive schemes are described below:

1. *Token-Exchange Model*: This is a hard incentive model. Users are given a set of tokens to begin with and with every file downloaded, they must pay one token to the file owner. Once they are out of tokens, they cannot download anymore files. Users can gain tokens by providing their files for sharing. Thus, users tend to advertise their files in order to gain tokens. This model makes users hesitate to download files since the cost is high. They must decide whether or not to spend their token on a given file.

2. *Peer-Approved Model*: In this model, users are given ratings and they are only allowed to download files from a user that has a lower or equal rating as them. This will thus encourage users to advertise their files and increase their ratings. User ratings can be based on the number of file-requests served or the number of files advertised by the user. First-time users that do not have files of their own are allowed to download a small number of files.
3. **Service-Quality Model**: This is similar to the Peer-Approved Model, but instead of restricting user downloads to be served by lower or equal reputation users, user requests are ranked into service levels according to the user’s rating.

The evaluation provided by the authors suggests that non-pricing schemes may be more practical to implement in certain kinds of collaborative networks than direct payments between users. Users may prefer (and thus accept more quickly) schemes that do not require payments or decisions for each transaction.

In the research carried out by Farzan et al. [13], we again see reward-based incentives coupled with reputation. The authors experiment with various incentive systems, as given below:

- **Incentivizing with Rewards**: Provide users a material reward for their contribution. This reward can be scaled according to the quality of the user’s participation instead of giving a fixed reward to all users regardless of the rating for their contribution by other users.

- **Incentivizing by Explaining Community Benefit**: Let users know the benefit they are causing to other users. Experiments with this model confirmed that people are more likely to work if they feel that their contribution is important and beneficial to others. This is similar to an altruistic incentive model. Moreover, it was found that letting the user know the exact beneficiary helps even more in motivating the user.

- **Incentivizing by Goal-Setting**: Provide users with intense and specific short-terms goals. The authors discover that this is more beneficial than providing them vague long-term goals when it comes to encouraging their participation. Furthermore, grouping goals are even more helpful in this respect. This can be seen in practice in many incentive-based applications, for example, LinkedIn [27] provides users with a progress bar of how complete their profile is, Fitbit [15] periodically reminds users how close they are to meeting their daily fitness goals, etc.
• **Incentivizing by Reputation:** Provide users with reputation points and ratings to improve their social standing. The authors find this to be the most beneficial when designing their incentive model. This kind of incentive mechanism has proven to be very useful in practice. For example, Flickr [17] addresses a user’s reputation by highlighting certain content on that user’s profile as “their most interesting photos”.

• **Incentivizing by Providing Self-Benefit:** Encourage users’ participation by turning their feedback into an activity that is important and meaningful to them. An example of this is using student ratings for courses to show them where they stand with respect to their career goals.

The authors conclude on using a point-based reputation system to provide incentives. Along with this, they highlight users with high contributions. They observe higher participation rates than before and also higher than what they had observed with just a reward-based system.

### 2.2.1 Altruism

Another interesting approach to incentivizing participation on resource sharing social networks is making users aware of the public good of their contributions. Leider et al. [26] provide an experiment in which they measure how likely people are willing to help other people depending on how close the other person is to them and depending on whether they will receive something in return. The authors perform a study in which they define 3 scenarios:

1. **Baseline Altruism:** Altruism towards complete strangers.

2. **Directed Altruism:** Altruism which places friends above strangers.

3. **Reward-Based Incentives:** A scenario in which users are likely to receive something for their services in future interactions.
The authors model situations in which a user would prefer to help his friend and thereby incur a cost rather than damage their friendship. The authors found that directed altruism is a much stronger (about twice as strong) factor than the prospect of future interactions and both are much stronger than baseline altruism. This suggests that people prefer to help their friends without expecting anything in return, but will not be willing to do so when it comes to strangers. People tend to expect something in future interactions with a higher probability when it comes to providing services to strangers. Another interesting result of this paper was that there is a tendency for friends to cluster other people by baseline altruism and thus it is possible that altruistic behavior can spread through the network, but this is only likely to spread across links that are between close friends.

2.3 Stable Marriage Problem

The Stable Marriage Problem is used to model various scenarios in which the goal is to find a stable matching between elements belonging to 2 different sets. The key idea here is the notion of a stable match, which requires that no 2 pairs of matched participants would be better off trading their respective partners. The theoretical contributions towards this algorithm were made by Lloyd Shapley and David Gale in their seminal paper D. Gale [11]. This theoretical work was applied to various scenarios by Alvin Roth, such as in matching hospitals and doctors, students and high-schools, kidneys and patients, etc. These various applications of the Gale-Shapley algorithm won Alvin Roth and Lloyd Shapley the Nobel Prize in Economics in 2012. In order to find a stable matching for the Stable Marriage Problem, participants belonging to both sets are asked to provide their preference lists for being matched with participants of the other set. Due to the generic nature of the inputs, this algorithm can easily be applied to a variety of scenarios. Section 3.4 talks more in detail about the working of the Gale-Shapley algorithm, as well as the variations to this algorithm which have been used when developing the matching algorithm used in Quet.
CHAPTER 3
QUET: THE ON-DEMAND PEER-TO-PEER KNOWLEDGE SHARING APP

In this chapter, we describe Quet, the mobile application that we have developed for on-demand peer-to-peer tutoring and knowledge sharing. We provide an overview of Quet in section 3.1. Section 3.2 provides a sample working of Quet with the example of 2 users being matched up. We describe Quet’s underlying incentive model in section 3.3. Section 3.4 describes how users are matched with the best possible peers. Finally, we describe Quet’s robust architecture in section 3.5.

3.1 Overview of Quet

In this section, we will provide an overview of Quet. We developed Quet as an application for our research on on-demand peer-to-peer tutoring and knowledge sharing. Quet is an Android app based on a sharing economy model which allows users to request help in their coursework on-demand. Once users submit their help-request, Quet uses their preferences, budget and course to match them up with the best available mentors. Quet also includes the preferences of the matched mentor, their price per help-request session and their course to make these matches. Once Quet has matched up users, they are expected to use the in-app chat to physically meet up and discuss the help-request. Users do not need to worry about finding the best help available themselves and the application is very easy to use. Section 3.4 describes how user preference lists are generated and how the matching algorithm works. In order to understand the basic working of Quet, we first introduce some basic terms and details of the application.
• **Quet Credits (QC):**
  Quet uses a virtual currency called “Quet Credits”, abbreviated as “QC”. These are units of the hard incentive which are transferred from matched mentees to their respective mentors. Users will initially be given a certain number of Quet Credits so they can start requesting help on Quet. All transactions on Quet which involve hard incentives make use of Quet Credits. The experimental setup explained in chapter 4 makes use of the total Quet Credits accumulated by users (excluding free sessions) in order to pay them at the end of the experimental period. Thus, Quet Credits can be thought of being synonymous to actual money.

• **Help-request sessions:**
  Users who wish to use Quet in order to request help and get matched with a peer, can do so by filling out a simple help-request form. Once a user submits his or her help-request, they are matched up by Quet to the best available mentor. A help-request session can be up to 30 minutes long.

• **Budget:**
  Each user when setting up their profile on Quet, specifies his or her budget when being matched up as a mentee. This is the amount of Quet Credits that they are willing to pay their matched mentors per help-request session. Quet ensures that a user requesting help never gets matched up with a peer whose price is more than the former’s budget. Quet allows only 3 distinct values for the budget; namely 0, 1, 2 QC. Users who wish to use the application to receive help for free may specify 0 QC as their budget. The reason we have resorted to keeping payments this low is that 2 QC (the maximum allowed transaction amount) is akin to the price of coffee (remember that QC are synonymous to actual money, or dollars in this case). This is based on the simple observation that a student receiving help from his or her peer would offer that peer a cup of coffee out of courtesy anyhow (at least they should). Quet just makes the arduous task of finding helpful peers easy.
• **Price:**
Each user when setting up their profile on Quet, specifies his or her price when
being matched up as a mentor. This is the amount of Quet Credits that they
expect to be paid by their matched mentees per help-request session. Quet
ensures that a user getting matched as a mentor never ends up getting paid
less than their price. Again, Quet allows only 3 distinct values for the price;
namely 0, 1, 2 QC. Users who wish to provide help to their peers for free may
specify 0 QC as their price.

• **Optimal budget and price values:**
Though there are no fixed optimal values for the budget and price that a user
can set, some obvious facts can be taken into consideration by users. Setting a
low budget will most likely hinder a user’s chances of getting matched up with
a mentor. Conversely, a high budget can increase their chances. Similarly, a
user who sets a low price will have a higher chance of getting matched up with
a mentee and one who sets a high price will have a low matching chance. Thus,
Quet prefers users with altruistic intentions.

• **Wallet:**
Each user has a wallet on Quet which specifies the total number of Quet Credits
they currently have available. This wallet changes depending on a user’s help-
request interactions with his or her peers. When being matched up as a mentor
with a peer, the user’s wallet is incremented by the payment amount of the
help-request session. Conversely, when being matched up as a mentee, the
corresponding payment amount is deducted from the user’s wallet. The method
used in deciding the payment of each help-request session has been described
below.

• **Mentor matching criteria:**
Users are asked to set up their matching criteria when being matched up with
someone as a mentor, as part of their profile information. This basically asks
users what kind of mentee they prefer getting matched up with. There are 5
available options for this criteria which range from preference being given to
the amount of QC earned (pareto_0), to the mentee’s rating (pareto_4). Figure
3.1 shows these possible options. As we will see in section 3.4, this matching
criteria is used to generate the preference list for each matched up mentor.

<table>
<thead>
<tr>
<th>Only care about QC earned</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prefer high QC earned over mentee rating</td>
</tr>
<tr>
<td>Care equally: QC earned &amp; mentee rating</td>
</tr>
<tr>
<td>Prefer high mentee rating over QC earned</td>
</tr>
<tr>
<td>Only care about mentee rating</td>
</tr>
</tbody>
</table>

Figure 3.1: Different possible options for the mentor matching criteria.

- **Mentee matching criteria:**
  Users are asked to set up their matching criteria when being matched up with
  someone as a mentee, as part of each help-request that they submit. This
  basically asks users what kind of mentor they prefer getting matched up with.
  There are 5 available options for this criteria which range from preference being
  given to the amount of QC spent per help-request session (pareto_0), to the
  mentor’s rating (pareto_4). Figure 3.2 shows these possible options. As we will
  see in section 3.4, this matching criteria is used to generate the preference list
  for each matched up mentee.

- **Payment per help-request session:**
  The payment amount for a help-request session between a matched mentor and
  mentee depends on their respective matching criteria. Quet already ensures
  that the matched mentor charges within the budget of the matched mentee.
To decide the actual amount that the mentee owes the mentor, Quet must find the most suitable amount dependent on both user’s matching criteria. For brevity, we will refer to the mentor’s mentor matching criteria simply as the mentor matching criteria and the mentee’s mentee matching criteria as the mentee matching criteria. We look at the mentor matching criteria and the mentee matching criteria to find the payment amount according to the matrix provided in figure 3.3. Note the following range for the payment of a help-request session:

\[
\text{payment} \in \{\text{mentor’s price, mentee’s budget}\}
\]

From this range, we can see that the mentor would prefer being paid an amount equivalent to the mentee’s budget since that is the highest that the payment for that help-request session can be. Conversely, the mentee would prefer being paid an amount equivalent to the mentor’s price since that is the lowest amount that the payment for that help-request session can be. From figure 3.3, we see that the mentor is favored whenever his or her matching criteria is closer to the first option i.e. \textit{pareto}0 than the mentee matching criteria and vice versa.
for cases that favor the mentee. When both mentor and mentee have the exact same matching criteria, we cannot favor one over the other and thus randomly choose the help-request session payment to be either the mentor’s price or the mentee’s budget.

- **Anonymous ratings:**
  At the end of each help-request session, the mentor rates his or her matched mentee and the mentee rates his or her matched mentor. These ratings are anonymous. Quet does not disclose user ratings at all; in fact, a user is not even shown his or her own rating. This is done to avoid awkwardness amongst peers since Quet is designed to be used within a classroom. Thus, each user on Quet has a list of mentor ratings (ratings given by their matched mentees when the user was a mentor to them) and a list of mentee ratings (ratings given by their matched mentors when the user was their mentee). Section 3.4 explains
how these mentor and mentee ratings are used to generate user preference lists in the matching algorithm.

3.2 Sample Working of Quet

In this section, we describe a sample working of Quet in order to explain a typical usage of the application. The case we cover is the general one, without any exceptions and assuming that neither user cancels the request. We will first describe parts of the application that are common to all users, including consent pages, the profile page, etc. For the remaining part of this section, we will explain Quet’s working with the help of 2 users interacting with each other. Chinmay will assume the role of a mentee by requesting help and Bill will assume the position of the mentor who gets matched up to Chinmay via Quet’s matching algorithm.

3.2.1 Application Working Common to All Users

In this section we describe parts of the application that all users see, irrespective of whether they are assuming the role of a mentor or mentee in a help-request session. We will go over screenshots of the application to help understand the different stages involved in these common functionalities.

1. In-app consent documents:
   On installing the application, users will be shown the landing page of Quet, shown in figure 3.4. Users will then be guided through a series of consent documents explaining them how their privacy is maintained on the application, their rights as participants in using Quet, some basic information about Quet, etc. We omit these screenshots for the sake of conciseness. Finally, users are guided towards the final page which asks them for their consent, as shown in figure 3.5. If the user chooses not to provide their consent, the application will be unusable and they will not be allowed to proceed in order to protect their
privacy, as shown in figure 3.6. On entering a valid Illinois email ID, users are sent a confirmation email via which they can set up their password to log in to Quet. A sample confirmation email is shown in figure 3.7.
Figure 3.5: Statement of consent page.

Figure 3.6: Error received if the user does not consent to use Quet.
2. Logging in and creating your profile:

After a user has consented to use Quet and has set up their password, they are shown the login page (see figure 3.8). If a user is logging in for the first time, they are prompted to create their Quet profile wherein they are asked to enter the courses for which they plan on using Quet, their budget, price and mentor matching criteria. As can be seen from the screenshots shown in figure 3.9, the user is given an idea of what their optimal budget and price values should be. Finally, once the user’s profile has been created, they can view their profile (Chinmay’s profile has been shown in figure 3.10). Note that the profile also shows the user’s current wallet. The user can choose to edit any aspect of his or her profile from the profile page by clicking the “Edit Profile” button.

3.2.2 Application Working Specific to Mentee

In this section we describe parts of the application that are specific to the mentee. A user requests help from his or her profile page. If an available mentor that is matched by Quet accepts the user’s help-request, they can use the in-app chat in order to decide when and where to meet up. Once the users have met, the mentee presses the “Start Session” button. After the mentor has finished helping the mentee, he or she presses the “End Session” button. Finally, both users are shown a rating page and then directed back to their respective profile pages. We will go over screenshots of
the application to help understand the different stages involved for user’s assuming the role of a mentee. In our walkthrough example, Chinmay is the mentee.

1. **Requesting help:**
   Users can submit a help-request by pressing the “Request Help” button on their profile page (see figure 3.10). Figure 3.11 shows screenshots of various pieces of information that users must enter before submitting their help request. In our example, Chinmay requests help for the question “How do you find the best split for decision trees?”. He enters the course that his question pertains to i.e. CS 412. He also enters his estimation for how long he thinks it may take for a matched up mentor to help him for this question. Finally, Chinmay also enters a location where he may be able to meet with the matched mentor, as well as his mentee matching criteria. Thus, the screenshots in figure 3.11 are of the mentee’s (Chinmay’s) phone.

2. **Possible responses to help-requests submitted:**
   A user that has submitted a help-request may receive many different responses.
(a) Basic “Edit Profile” page.

(b) Specify budget.

(c) Specify price.

(d) Specify mentor matching criteria.

Figure 3.9: Editing your profile.
A brief explanation of possible responses has been provided below.

(a) *Can’t be matched:* In case Quet is unable to find the help-requester a peer to match with as a mentor, it will send the help-requester a notification indicating a failure to match his or her help-request. This may be due to a number of reasons such as the lack of available peers who can be matched as potential mentors (users currently engaged in a help-request session are considered unavailable), a low budget set by the help-requester (note that Quet will always match a user with a mentor whose price is within the user’s budget, thus setting a low budget could reduce the user’s chances of getting matched), etc.

(b) *Can’t find a mentor to help for the specific question:* In case Quet matches the help-requester to a mentor who declines the help-request, it will send the help-requester a notification specifying this. Note that this notification does not reveal information about the matched mentor who declined the user’s help-request.
Figure 3.11: Making a help-request.
(c) *Help-request has timed out:* In case Quet matches a help-requester to a mentor who does not accept or decline the help-request within 15 minutes, it will send the help-requester a notification specifying that his or her help-request has timed out. Quet provides matched mentors a time window of 15 minutes to respond to a help-request that they have been matched up with. Failure to respond (either accept or decline) to the help-request will lead to this notification being sent to the help-requester.

(d) *Found a mentor:* In case Quet matches the help-requester to a mentor who accepts the help-request, it will send a notification to the help-requester specifying this. In our sample working, Quet matches Chinmay’s help-request to Bill and sends him a notification. Considering that Bill accepts Chinmay’s help-request, Chinmay receives a request-acceptance notification and his screen changes as shown in figure 3.12. Chinmay now sees Bill’s name and display picture on his profile indicating that Bill has accepted his help-request. Information about the accepted help-request is also shown. Note that both users are now able to use Quet’s in-app chat to decide when and where to meet up in order to start their help-request session.

3. **Chatting with the matched mentor:**

   Once the matched mentor has accepted the help-requester’s help-request; the help-requester assumes the role of the mentee. The mentor and mentee may now use Quet’s in-app chat to decide when and where to physically meet up. In our sample working, Bill accepts Chinmay’s help-request. Chinmay and Bill now use the in-app chat as shown in figure 3.13. The screenshot represents the mentee’s (Chinmay’s) view of the in-app chat and that is the reason why information about the matched mentor (Bill) is seen on the screen.
4. **Cancelling the help-request after the matched mentor has accepted it:**

Both matched users have the option of cancelling the help-request session even after the matched mentor has accepted it. This option has been kept in order to handle emergency situations and cases wherein the matched users are unable to meet up in the near future. Note that this can only be done before the users have started the help-request session (initiated by the matched mentee pressing the “Start Session” button). Thus, we can see that the mentee (Chinmay) sees a “Cancel” button at the bottom of his screen in figure 3.13. A similar “Cancel” button can also be seen on the mentor’s (Bill’s) screen. Pressing the cancel button triggers a notification to be sent to the matched user informing him or her that their match has cancelled the help-request.
5. **Starting the help-request session:**

   Once both users decide a time and place to meet up and have physically met up, the matched mentee presses the “Start Session” button. In our sample working, Chinmay’s screen shows a “Start Session” button as can be seen in figure 3.13. Once Chinmay and Bill meet up, Chinmay must press this button in order to initiate the start of the help-request session.

6. **Help-request session time up notification:**

   Once the matched mentee has pressed the “Start Session” button, Quet will send both matched users a reminder notification after the entered session time has run out. In our sample working, Chinmay had submitted a help-request of 20 minutes (refer to figure 3.11), thus Quet will send Chinmay and Bill a “time-up” reminder notification 20 minutes after Chinmay presses “Start Session”. This notification only serves the purpose of a reminder and does not force either users out of the help-request session. Users may continue to be in a session for
as long as they wish and until the matched mentor presses the “End Session” button. Note that the matched mentor will only be paid for the original session and not for the extra time used up after the session has timed up.

7. Rating the matched mentor:
   Once the matched mentor has finished helping the mentee satisfactorily, he or she must press the “End Session” to indicate that their help-request session has finished. Once the mentor presses the “End Session” button, both users see a rating page wherein they must rate their matched peer according to their experience. Figure 3.14 shows what Chinmay (the mentee) sees on his screen after Bill (the mentor) has ended their help-request session. The mentee may decide what to rate his or her matched mentor depending on how satisfactory the mentor’s help was, how the mentor’s attitude was towards providing help, etc.

Figure 3.14: Chinmay sees the rating page to rate his mentor Bill.
3.2.3 Application Working Specific to Mentor

In this section we describe parts of the application that are specific to the mentor. A user may receive a notification from Quet indicating that he or she has been matched with a help-request. This notification is accompanied by information related to the help-request (excluding the name and display picture of the help-requester). A matched mentor has 15 minutes to react to this matched help-request. The user can choose to decline the request in which case the user is directed back to his or her profile page. In case the user feels that he or she can provide adequate help for the matched help-request, the user should accept the help-request. If a user accepts or declines the help-request after the 15-minute response time, he or she is directed back to their profile page, since the help-request has timed out. A user that accepts a help-request will assume the position of the matched mentor and can use Quet’s in-app chat to decide when and where to meet up with his or her matched mentee. Once the users have met, the mentee presses the “Start Session” button. After the mentor has finished helping the mentee, he or she presses the “End Session” button. Finally, both users are shown a rating page and then directed back to their respective profile pages. We will go over screenshots of the application to help understand the different stages involved for user’s assuming the role of a mentor. In our walkthrough example, Bill is the mentor.

1. **Getting matched as a mentor:**
   
   If a user gets matched up as a mentor for a particular help-request, he or she receives a notification from Quet. Along with the notification, the user is shown information about the help-request. This includes the question, the course it pertains to, the estimated time for the help-request (as entered by the help-requester), the payment they would receive if they accept the help-request and the location entered by the help-requester. Note that the user that has been matched as a mentor does not see the help-requester’s information (display name and profile picture), until the matched user accepts the help-request. This is done to prevent users from deciding to accept or decline help-requests depending on who asks for help. If the matched user declines the help-request,
he or she is directed back to his or her profile page. As mentioned previously, a matched user has 15 minutes to respond to a help-request. If the user responds (accepts or declines) after this stipulated time, a “request timed out” error is shown to the user and the user is directed back to his or her profile page. Figure 3.15 shows the notification containing the payment information and the screen with the help-request information, as seen on the mentor’s (Bill’s) phone.

(a) The notification sent to Bill after Quet matches him up as a mentor for Chinmay’s help-request. Note that the payment for the help-request is mentioned in the notification.

(b) Change in Bill’s screen after being matched up as a mentor. Note that information about Chinmay is not yet visible to Bill.

Figure 3.15: The match notification and screen change once Chinmay’s help-request has been matched up to Bill.
2. **Accepting the help-request:**

   If the matched up user accepts the help-request before it times out, he or she assumes the role of as mentor and is able to see the mentee’s information. The mentor and mentee are also able to chat with each other. In our sample working, Bill accepts Chinmay’s help-request and is thus able to see Chinmay’s information on the previous screen that showed him information about the help-request (see figure 3.16).

![Figure 3.16: Bill accepts Chinmay’s help-request and is thus able to see Chinmay’s information and chat with him.](image)

3. **Chatting with the matched mentee:**

   After accepting the help-request, the matched mentor and mentee can use the in-app chat to decide when and where to meet up. Figure 3.17 shows us what Bill’s screen looks like while he is chatting with his matched mentee, Chinmay.
4. **Cancelling the help-request:**

   A mentor may cancel his or her matched help-request even though he or she has already accepted it. This may be done until the matched mentee starts the help-request session.

5. **Screen change after mentee starts the help-request session:**

   Once the matched mentor and mentee have physically met up; the mentee presses the “Start Session” button. As soon as the “Start Session” button is pressed, the mentor’s screen changes such that the “Cancel” button is replaced by an “End Session” button. Figure 3.18 shows the change in Bill’s screen after Chinmay presses the “Start Session” button. Similarly, once the mentee presses the “Start Session” button, the “Cancel” button will be removed from his or her screen.
6. Ending the help-request session:

Once the mentor has finished helping the mentee, he or she presses the “End Session” button. If the help-request session has not ended within the estimated time submitted by the mentee while requesting help (20 minutes in our sample working example), Quet will send “time up” reminder notifications to both the matched mentor and mentee. Once the mentor has ended the session, both users see the rating page. In our sample working, the mentor (Bill) sees the screen as shown in figure 3.19.

We describe the overall working of a typical interaction between 2 users matched up on Quet in the flowchart given in figure 3.20. This flowchart describes the interaction between the mentor Bill and the mentee Chinmay in our sample working.
Figure 3.19: Bill sees the rating page to rate Chinmay.

Figure 3.20: Flowchart describing the help-request session between Bill and Chinmay.
3.3 Incentive Model

Chapter 2 provides us with a detail description of existing research on incentive models. Some key points based on these research papers that were considered while designing the incentive model in Quet have been explained below.

- **Importance of reputation-based incentives:** Reputation-based incentive models tend to be more effective than ones that provide direct rewards, like money. The reason for this is the hesitation that users face while spending money, which can prevent them from seeking services. The producers of these services have in fact found to have a nearly equal level of participation irrespective of whether the incentive is reputation-based or money-based.

- **Adaptive incentive models:** Platform-centric incentives must adapt according to the current demand-supply trends in the system. Though Quet does not use platform-centric incentives, it implicitly increases the probability of high-charging users to be matched up as mentors if the demand for help increases.

- **Combining soft and hard incentives:** A lot of research suggests using a combination of soft and hard incentives. Soft incentives involve increased access to information, increased social reputation, increased access to services on the platform, etc. Hard incentives are generally monetary in nature.

- **Gamification:** Users can be given short-term goals to achieve, for example: Help $x$ users this week, where $x$ depends on the current level of demand. Another approach to achieving this is making users aware of highly active participants on the platform. This in turn can motivate them to increase their own level of participation.

- **Appreciating active users:** Intermittently broadcasting users that provide regular services to other users on the platform has found to be a very effective means to maintaining a high level of participation.
• **Social Exchange Theory:** Matching users up with mentors who they gave high ratings in their previous interactions with a higher probability, is likely to be more beneficial to them. This observation is in line with the Social Exchange Theory.

• **Reputation proportional to quality of service:** Make the reputation points that a user gets from providing his or her services proportional to the rating received by his or her matched up peer.

Based on the above observations, we have devised an incentive model specific to the classroom context in which Quet is designed to be deployed. The incentive model implemented within Quet has the following salient features:

• **Hard incentives:**
  Users set their price as part of their Quet profile. When being matched up as a mentor for a peer, the user is guaranteed at least as much as their price, as the payment for each specific help-request session. The payment received serves as a (hard) incentive for users.

• **Soft incentives:**
  In order to incentivize participation via social reputation, Quet sends out broadcast notifications to all users informing them of the highest rated mentors. The actual mentor ratings of the mentors are not revealed, but their names are shown in order to motivate users that value social reputation.

• **Altruism:**
  Quet allows users to set a zero price for providing help. Not only will this motivate users who wish to provide help out of altruism, but will also increase the chances of altruistic users getting matched up as mentors. Section 3.4 describes how feasible candidates are chosen for each user before ranking them according to the user’s preferences. This candidate list generation phase is dependent on the price of a candidate mentor and the budget of a candidate mentee, apart from other factors.
• **Gamification and appreciation to increase motivation:**
  Broadcasting top-rated mentors not only serves as a direct soft incentive for users to participate, but also draws upon the studies of incentive mechanisms described previously, to use user appreciation to gamify participation. We hope to see a spirit of friendly competition amongst users to help as many peers as possible. By broadcasting user names, users who see a friend’s name as a top-rated mentor may be motivated to also perform better and help as many peers as they can, in hopes of getting their name up on the top-rated mentor’s list.

• **Preferential matching of candidates providing high quality of service:**
  Quet makes use of user ratings to decide the best possible matches, apart from other factors (explained in section 3.4). Though these ratings are anonymous, Quet’s matching algorithm makes use of these ratings in the background in order to sort candidate matches. In general, users with high mentor ratings will have a higher likelihood of appearing at the top of their peer’s preference lists than users with low mentor ratings.

• **Preferential matching of low cost mentors:**
  Users who set low price values will be within the budget of a larger number of peers than users who set high price values. By ensuring that a help-requester (possible mentee) never pays more than his or her budget, Quet enforces a probable mentor user to be feasible only if his or her price is within the help-requester’s budget.

• **Preferential matching of high budget mentees:**
  Users who set high budget values will be feasible candidates for a larger number of peer mentors than users who set low budget values. This is the converse of the above-mentioned point and is due to Quet’s restriction on help-request payments being at least as much as the price of the matched mentor.
• **Avoiding the free rider problem:**
  New users are initially provided with some initial Quet Credits in order to get started with requesting help on Quet. Once this initial amount has been consumed, users are required to help their peers (in order to potentially earn Quet Credits), before they are allowed to request help again. This is done to avoid free riding on the application and to prevent users from using the application just for requesting help and never providing help themselves. Of course, a user who has 0 QC in his or her wallet and a 0 budget can still potentially be matched up with an altruistic peer (one whose price is 0 QC), but as mentioned previously, the chances of such a user being matched is low.

Thus, the incentive model implemented in Quet makes use of a combination of virtual currency (translated to money) and social reputation in order to motivate users to contribute on this platform.

### 3.4 Matching Algorithm

In this chapter, we will provide an overview of the matching algorithm used within Quet to find the best possible matches of mentors to mentees. We first describe the methods used to generate preference lists for each user, wherein feasible candidates are ranked for each user. We later describe the variant of the Gale-Shapley algorithm used to find the best possible matches.

As mentioned in section 3.1, each user has a list of mentor ratings and a list of mentee ratings. Throughout this chapter, whenever we mention “mentor’s rating”, we are referring to the user’s average mentor rating. Similarly, whenever we mention “mentee’s rating”, we are referring to the user’s average mentee rating. Note that users are not matched immediately as they submit their help-requests. The matching algorithm is run every 20 minutes and users that had submitted their help-requests within this matching cycle are matched to the best available mentors according to the preferences of both the user and the matched mentor. This waiting period for each matching cycle is essential in order to obtain the best possible mentor-mentee matches.
according to the preference lists generated for both types of users and according to
the output of our variant of the Gale-Shapley algorithm.

3.4.1 Generating User Preference Lists

At the beginning of each matching cycle, the help-requests submitted in that cycle
are retrieved. In this section, we first describe the process of finding a set of feasible
mentors for each help-request submitted. Similarly, we also populate a set of feasible
mentees for each of these feasible mentors. Later, we explain how these feasible
candidates are sorted according to user preferences.

3.4.1.1 Populating the Feasible Candidate Lists

The procedure used in populating feasible candidate lists for help-requesters and
each of their feasible candidate mentors is explained below.

1. For every help-request that is submitted, we find all candidate mentors that
are feasible for that help-request.

2. Since a help-requester can specify a different mentee matching criteria with
every help-request, each help-request should be considered as a separate candi-
date to be matched with an available mentor. Thus, if a user submits multiple
help-requests within a single matching cycle, each help-request is considered
as a separate candidate. Each candidate (called a help-request object from
now on) is implemented as a tuple of mentee user_id and the question_id of the
help-request.

3. Note that users that have requested help in a matching cycle are not considered
as candidate mentors for that matching cycle since a user expecting help may
not appreciate being matched as a mentor with someone else, before having
received help first.
4. Feasible mentors for a given help-request are found using the budget of the help-requester, the course that the help-request pertains to, the price of the candidate mentor user and the courses that the candidate mentor user has signed up for.

5. Feasible mentors for a help-request are those users who have signed up for the course mentioned in the help-request and who charge within the budget of the help-requester.

6. Feasible mentors for each help-request are populated. Thus, we generate a dictionary mapping from help-request object to a list of feasible mentors.

7. We can use this dictionary mapping to create a reverse mapping for each of the feasible mentors. We thus populate each help-request object whose list of feasible mentors contains the specified user. The list of feasible mentees for each available user thus consists of all the help-request objects for which this user is a feasible mentor.

8. Note that we call this a feasible “mentee” list, though this is actually a feasible “help-request object” list. This is done for the sake of simplicity. Thus, we generate a dictionary mapping from mentor to a list of help-request objects.

3.4.1.2 Obtaining Pareto Frontier Candidates

In the previous subsection, we saw how to generate the feasible candidate lists. The 2 sets of feasible candidate lists obtained i.e. (help-request object to list of feasible mentors; and the mentor to list of feasible help-request objects) are unsorted. In this subsection, we describe the generic algorithm for finding candidates that lie on the Pareto frontier and those that have been dominated. When finding Pareto frontier mentors (for each help-request object), we compare candidates based on their price (x-axis) and their mentor rating (y-axis). Similarly, when finding Pareto frontier mentees (for each mentor), we compare candidates based on their budget (x-axis) and their mentee rating (y-axis).
Algorithm 1 goes over the pseudocode for the function $GetParetoFrontier$, which is used for retrieving candidates lying on the Pareto frontier and candidates that have been dominated by these Pareto frontier candidates. This function can be used both in the case of mentees and mentors. The function $GetParetoFrontier$ is given a list of feasible candidates sorted according to the price (cheapest first) if the candidates are mentors, and budget (highest paying first) if the candidates are mentees. The algorithm starts off with that candidate who has the most desirable (x-axis) attribute i.e. the cheapest mentor or the highest paying mentee. We then iterate over the remaining candidates, subsequently adding them to the list of Pareto frontier candidates if their rating is at least as much as the previously added Pareto frontier candidate. If the current candidate’s rating is lower than the previously added Pareto frontier candidate, this candidate is clearly a poor choice as compared to the previously added Pareto frontier candidate since it has both a less desirable (x-axis) attribute and also has a lower rating. Specifically, a dominated mentor candidate would be one who is more expensive than the previously added Pareto frontier mentor and who also has a lower mentor rating than him or her. Similarly, a dominated mentee candidate would be one whose budget is lower than the previously added Pareto frontier mentee and who also has a lower mentee rating than him or her.
Algorithm 1 Finding the candidates lying on the Pareto frontier and those that have been dominated.

1: function GetParetoFrontier(l) \(\triangleright \) "l": list of feasible candidates sorted according to the price (ascending) for mentors and budget (descending) for mentees.
2: \(P \leftarrow l[0]\) \(\triangleright \) List to store Pareto frontier candidates.
3: \(D \leftarrow [ ]\) \(\triangleright \) List to store dominated candidates.
4: for each \(x \in l[1 : ]\) do
5: \(\) front \(\leftarrow PgetProperty()
6: \(\) if \(x.rating \geq \) front.rating then
7: \(\) \(P \leftarrow P + [x]\) \(\triangleright \) Append the current candidate to the Pareto frontier candidates.
8: \(\) else
9: \(\) \(D \leftarrow D + [x]\) \(\triangleright \) Append the current candidate to the list of dominated candidates.
10: \(\) end if
11: end for \(\) return \(P, D\)
12: end function

- Obtaining Pareto frontier candidates for mentees:
  In order to find mentors from a given help-request object’s feasible mentor list, who are either part of the Pareto frontier or are dominated candidates for that help-request object, we call the function GetParetoFrontier described in algorithm 1, passing this list of feasible mentors sorted by their price (cheapest first). In this implementation, we use the mentor’s ratings as the candidate ratings. Since we are analyzing this for the case of mentees, we must consider a trade-off between a mentor’s price (x-axis) and the mentor’s rating (y-axis). Feasible mentors lying on the Pareto frontier for a mentee are those mentors that are all Pareto efficient. Mentors that are dominated by these Pareto frontier candidates are those that are more expensive than the previously added mentor, despite being lower rated than them.
Figure 3.21 shows an example of how Pareto frontier candidates and dominated candidates might be distributed for a given mentee. Note that the feasible mentors can have a price only up to the mentee’s budget. The mentor rating (y-axis) is the average mentor rating received by the feasible mentors over their past help-request sessions wherein they were mentors. The sorted candidate list passed to $GetParetoFrontier$ in this case would be $[A, B, C, D, E, F, G, H, I, J]$. Note that for mentors with the same price, we prefer the higher rated mentor when sorting. The $P$ list (containing candidate mentors on the Pareto frontier) that we obtain is: $[A, B, D, F, G, I]$ and the dominated candidate list is: $[C, E, H, J]$.

![Figure 3.21: Example of mentors lying on the Pareto frontier and those that are dominated for a given help-request object.](image)

- **Obtaining Pareto frontier candidates for mentors:**
  In order to find mentees (corresponding to help-request objects) from a given mentor’s feasible help-request object list, who are either part of the Pareto frontier or are dominated candidates for that mentor, we call the function $GetParetoFrontier$ described in algorithm 1, passing this list of feasible help-request objects sorted by their mentee’s budget (highest paying first). In this
implementation, we use the mentee’s ratings as the candidate ratings. Since we are analyzing this for the case of mentors, we must consider a trade-off between a mentee’s budget (x-axis) and the mentee’s rating (y-axis). Feasible mentees lying on the Pareto frontier for a mentor are those mentees that are all Pareto efficient. Mentees that are dominated by these Pareto frontier candidates are those with lower budgets and lower ratings than the previously added mentee. Figure 3.22 shows an example of how Pareto frontier candidates and dominated candidates might be distributed for a given mentor. Note that the feasible mentees can only pay as low as the mentor’s price. The mentee rating (y-axis) is the average mentee rating received by the feasible mentees over their past help-request sessions wherein they were mentees. The sorted candidate list passed to GetParetoFrontier in this case would be \([A, B, C, D, E, F, G, H, I, J]\). Note that for mentees with the same budget, we prefer the higher rated mentee when sorting. The \(P\) list (containing candidate mentees on the Pareto frontier) that we obtain is: \([A, B, D, F, G, I]\) and the dominated candidate list is: \([C, E, H, J]\).

![Figure 3.22: Example of mentees lying on the Pareto frontier and those that are dominated for a given mentor.](image)

An understanding of candidates on the Pareto frontier is essential to explain how preference lists are generated from the list of feasible candidates. In the following
subsubsections, we describe the methods used to sort feasible candidate lists according to the mentee and mentor matching criteria respectively.

3.4.1.3 **Sorting Feasible Mentors According to the Mentee Matching Criteria**

In this subsection, we describe how the list of feasible mentors pertaining to each help-request object is sorted according to the mentee matching criteria. Note that a user supplies a matching criteria when submitting each help-request. We have described the different possible options for the mentee matching criteria in section 3.1. Algorithm 2 goes over the pseudocode for retrieving a sorted list of feasible mentors depending on the corresponding help-request object’s mentee matching criteria.
Algorithm 2 Sorting the list of feasible mentors according to the mentee matching criteria.

1: function SortFeasibleMentors(c, l) \> “c”: mentee matching criteria, 
\> “l”: list of feasible mentors.
2: \hspace{1em} if $c = \text{pareto.4}$ then
3: \hspace{2em} $S \leftarrow \text{sorted}(l, \text{key} = [(\text{mentor ratings, -}), (\text{price, +})])$
4: \hspace{1em} else
5: \hspace{2em} $S_{\text{Price}} \leftarrow \text{sorted}(l, \text{key} = [(\text{price, +}), (\text{mentor ratings, -})])$
6: \hspace{2em} if $c = \text{pareto.0}$ then
7: \hspace{3em} $S \leftarrow S_{\text{Price}}$
8: \hspace{1em} else \> \> Using $S_{\text{Price}}$ to sort based on Pareto frontier candidates.
9: \hspace{2em} $P, D \leftarrow \text{GetParetoFrontier}(S_{\text{Price}})$
10: \hspace{2em} if $c = \text{pareto.1}$ then
11: \hspace{3em} $S \leftarrow P + D$
12: \hspace{2em} else if $c = \text{pareto.2}$ then
13: \hspace{3em} $S \leftarrow \text{Shuffle}(P) + \text{Shuffle}(D)$
14: \hspace{2em} else
15: \hspace{3em} $S \leftarrow P.\text{Reverse}() + \text{sorted}(D, \text{key} = [(\text{mentor ratings, -})])$
16: \hspace{2em} end if
17: \hspace{2em} end if
18: \hspace{1em} end if
19: return $S$

The SortFeasibleMentors function takes the mentee matching criteria and the mentee’s list of feasible mentors as its arguments. We provide a brief explanation of its working, below.

- **Pareto.4:** Only care about the mentor’s rating:
  
  If the mentee matching criteria is $\text{pareto.4}$, we only care about the mentor’s rating and thus we sort the list of feasible mentors according to their mentor rating in descending order. Note on line 3 that, although we only care about the mentor rating, we also apply the mentor’s price as the second key for sorting
This is done to break ties. Thus, we would still choose the cheaper of 2 mentors with equally high mentor ratings, in spite of the mentee matching criteria being pareto_4.

- **Sorting by mentor’s price:**
  For all other mentee matching criteria, we must sort the feasible mentors according to their price (cheapest first). Again we note on line 5 that we use both keys for sorting, in order to break ties.

- **Pareto_0: Only care about the QC spent:**
  If the mentee chooses this matching criteria, we wish to find the cheapest possible mentor, without caring about the mentor’s rating. As we have already sorted the feasible mentors according to their price (cheapest first) on line 5, this case simply requires us to return this list (see line 7).

- **Pareto frontier and dominated candidates:**
  For mentee matching criteria pareto_1, pareto_2 and pareto_3, we must first obtain a list of feasible mentors lying on the Pareto frontier (referenced on line 9 as P) and a list of feasible mentors that have been dominated by these Pareto frontier candidates (referenced on line 9 as D). We have described the working of the GetParetoFrontier function in algorithm 1). An interesting thing to note is that there is no notion of preference amongst the Pareto frontier candidates unless we specify which dimension is important to us i.e. they have equal utility. The same goes for dominated candidates. Thus, depending on the mentee matching criteria, (either pareto_1, pareto_2, or pareto_3), we are able to sort candidates within these respective lists.

- **Pareto_1: Prefer low QC spent over mentor’s rating:**
  If the mentee matching criteria is pareto_1, we give higher preference to finding a mentor who charges low than to a mentor who is highly rated. Note however that we are not completely indifferent towards the mentor’s rating. Candidates in the P (Pareto frontier) list obtained from the GetParetoFrontier function (see algorithm 1) are already sorted according to their price (cheapest...
to most expensive). Similarly, candidates in the $D$ (dominated candidates) list obtained from the $GetParetoFrontier$ function (see algorithm 1) are also already sorted according to their price (cheapest to most expensive). In order to give importance to mentor ratings, we must prefer Pareto frontier candidates over dominated candidates. The preference list in this case (see line 11) is the list of candidates lying on the Pareto frontier appended with the list of dominated candidates, thus giving a higher preference to Pareto frontier candidates than dominated candidates.

- **Pareto_2: Care equally about the QC spent and the mentor’s rating:** If the mentee matching criteria is $pareto_2$, we give no preference to either the mentor’s price or the mentor’s rating. We do, however, wish to give a higher preference towards candidates lying on the Pareto frontier than to those that have been dominated. There is no need to rank candidates within either lists according to either price or rating, though. We can see on line 13, that in order to achieve this, we simply randomly shuffle candidates belonging to the Pareto frontier and append that preference list to a randomly shuffled list of dominated candidates. This ensures that Pareto frontier candidates are preferred over dominated candidates, without ordering candidates within either lists according to any criteria.

- **Pareto_3: Prefer high mentor rating over the QC spent:** If the mentee matching criteria is $pareto_3$, we wish to give a higher preference to the mentor’s rating over his or her price. Note that the Pareto frontier list of candidates ($P$) that we obtain from the $GetParetoFrontier$ function (see algorithm 1) goes from lowest rated Pareto frontier candidate to highest rated Pareto frontier candidate. Thus, simply reversing this list produces the desired ordering amongst the Pareto frontier candidates. The dominated candidates list ($D$), however is not guaranteed to be reverse sorted according to mentor rating, thus we must explicitly sort this list so that higher rated dominated candidate mentors are given preference over lower rated dominated candidate mentors. Again, we still prefer Pareto frontier candidates over dominated can-
didates and thus append the sorted dominated candidates to the (reverse) list of Pareto frontier candidates, as can be seen on line 15.

- **Preference lists obtained from figure 3.21:**

Figure 3.21 shows an example of feasible mentors for a given help-request object. The mentor preference lists obtained for different mentee matching criteria in this specific example are provided in figure 3.23. We can see how drastically the preference list can change depending on the matching criteria.

<table>
<thead>
<tr>
<th>Preference lists for different mentee matching criteria:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Pareto.0:</strong> [A, B, C, D, E, F, G, H, I, J]</td>
</tr>
<tr>
<td><strong>Pareto.1:</strong> [A, B, C, D, E, F, G, I, H, J]</td>
</tr>
<tr>
<td><strong>Pareto.2:</strong> [A, B, D, E, F, G, I, H, J].Shuffle() + [C, E, H, J].Shuffle()</td>
</tr>
<tr>
<td><strong>Pareto.3:</strong> [I, G, F, D, B, A, J, H, E, C]</td>
</tr>
<tr>
<td><strong>Pareto.4:</strong> [I, G, J, F, H, B, D, E, A, C]</td>
</tr>
</tbody>
</table>

Figure 3.23: Preference lists obtained for different mentee matching criteria for candidates shown in figure 3.21.

### 3.4.1.4 Sorting Feasible Mentees According to the Mentor Matching Criteria

In this subsection, we describe how the list of feasible help-request objects (pertaining to each help-request submitted by a probable mentee) is sorted according to the mentor matching criteria. Note that a user sets the mentor matching criteria while setting up his or her Quet profile. We have described the different possible options for the mentor matching criteria in section 3.1. Algorithm 3 goes over the pseudocode for retrieving a sorted list of feasible mentees (each mentee corresponding to their respective help-request object), depending on the mentor matching criteria.
Algorithm 3 Sorting the list of feasible mentees according to the mentor matching criteria.

1: function SortFeasibleMentees(c, l) ▷ “c”: mentor matching criteria, “l”: list of feasible mentees.
2:     if c = pareto.4 then
3:         S ← sorted(l, key = [(mentee ratings, -), (budget, -)])
4:     else
5:         S_Budget ← sorted(l, key = [(budget, -), (mentee ratings, -)])
6:             if c = pareto.0 then
7:                 S ← S_Budget
8:             else ▷ Using S_Budget to sort based on Pareto frontier candidates.
9:                 P, D ← GetParetoFrontier(S_Budget)
10:                if c = pareto.1 then
11:                    S ← P + D
12:                else if c = pareto.2 then
13:                    S ← Shuffle(P) + Shuffle(D)
14:                else
15:                    S ← P.Reverse() + sorted(D, key = [(mentee ratings, -)])
16:                end if
17:             end if
18:     end if
19: return S

The SortFeasibleMentees function takes the mentor matching criteria and the mentor’s list of feasible mentees as its arguments. We provide a brief explanation of its working, below.

- **Pareto.4: Only care about the mentee’s rating:**
  If the mentor matching criteria is pareto.4, we only care about the mentee’s rating and thus we sort the list of feasible mentees according to their mentee rating in descending order. Note on line 3 that, though we only care about the mentee rating, we also apply the mentee’s budget as the second key for sorting.
(in descending order). This is done to break ties. Thus, we would still choose the higher paying of 2 mentees with equally high mentee ratings, in spite of the mentor matching criteria being pareto_4.

- **Sorting by mentee’s budget:**
  For all other mentor matching criteria, we must sort the feasible mentees according to their budget (highest paying first). Again we note on line 5 that we use both keys for sorting, in order to break ties.

- **Pareto_0: Only care about the QC earned:**
  If the mentor chooses this matching criteria, we wish to find the highest paying mentee, without caring about the mentee’s rating. As we have already sorted the feasible mentees according to their budget (highest paying first) on line 5, this case simply requires us to return this list (see line 7).

- **Pareto frontier and dominated candidates:**
  For mentor matching criteria pareto_1, pareto_2 and pareto_3, we must first obtain a list of feasible mentees lying on the Pareto frontier (referenced on line 9 as $P$) and a list of feasible mentees that have been dominated by these Pareto frontier candidates (referenced on line 9 as $D$). We have described the working of the GetParetoFrontier function in algorithm 1. As explained in the previous subsection, there is no notion of preference amongst the Pareto frontier candidates unless we specify which dimension is important to us i.e. they have equal utility. The same goes for dominated candidates. Thus, depending on the mentor matching criteria, (either pareto_1, pareto_2, or pareto_3), we are able to sort candidates within these respective lists.

- **Pareto_1: Prefer high QC earned over mentee’s rating:**
  If the mentor matching criteria is pareto_1, we give higher preference to finding a mentee who has a high budget than to a mentee who is highly rated. Note however that we are not completely indifferent towards the mentee’s rating. Candidates in the $P$ (Pareto frontier) list obtained from the GetParetoFrontier function (see algorithm 1) are already sorted according to their budget (highest
budget to lowest budget). Similarly, candidates in the \( D \) (dominated candidates) list obtained from the \textit{GetParetoFrontier} function (see algorithm 1) are also already sorted according to their budget (highest budget to lowest budget). In order to give importance to mentee ratings, we must prefer Pareto frontier candidates over dominated candidates. The preference list in this case (see line 11) is the list of candidates lying on the Pareto frontier appended with the list of dominated candidates, thus giving a higher preference to Pareto frontier candidates than dominated candidates.

- **Pareto.2: Care equally about the QC earned and the mentee’s rating:**
  If the mentor matching criteria is \textit{pareto}.2, we give no preference to either the mentee’s budget or the mentee’s rating. We do, however, wish to give a higher preference towards candidates lying on the Pareto frontier than to those that have been dominated. There is no need to rank candidates within either lists according to either budget or rating, though. We can see on line 13, that in order to achieve this, we simply randomly shuffle candidates belonging to the Pareto frontier and append that preference list to a randomly shuffled list of dominated candidates. This ensures that Pareto frontier candidates are preferred over dominated candidates, without ordering candidates within either lists according to any criteria.

- **Pareto.3: Prefer high mentee rating over the QC earned:**
  If the mentor matching criteria is \textit{pareto}.3, we wish to give a higher preference to the mentee’s rating over his or her budget. Note that the Pareto frontier list of candidates (\( P \)) that we obtain from the \textit{GetParetoFrontier} function (see algorithm 1) goes from lowest rated Pareto frontier candidate to highest rated Pareto frontier candidate. Thus, simply reversing this list produces the desired ordering amongst the Pareto frontier candidates. The dominated candidates list (\( D \)), however is not guaranteed to be reverse sorted according to mentee rating, thus we must explicitly sort this list so that higher rated dominated candidate mentees are given preference over lower rated dominated candidate mentees. Again, we still prefer Pareto frontier candidates over dominated can-
didates and thus append the sorted dominated candidates to the (reverse) list of Pareto frontier candidates, as can be seen on line 15.

- **Preference lists obtained from figure 3.22:**

  Figure 3.22 shows an example of feasible mentees for a given mentor. The mentee preference lists obtained for different mentor matching criteria in this specific example are provided in figure 3.24. We can see how drastically the preference list can change depending on the matching criteria.

  ![Preference lists obtained for different mentor matching criteria for candidates shown in figure 3.22.](image)

  Figure 3.24: Preference lists obtained for different mentor matching criteria for candidates shown in figure 3.22.

### 3.4.2 Finding Stable Matches for Mentors and Mentees

In this subsection we describe the stable marriage problem and the Gale-Shapley algorithm to find stable matches. We also describe our variation of the algorithm for unequal sets. Given 2 sets of elements of equal size with an ordered preference list for each element, the stable marriage problem deals with finding a stable matching between elements belonging to the 2 different sets.

**Stable Matching:** A matching is said to be stable if it exhibits Pareto efficiency. This ensures that there does not exist any match \((x, y)\) which both \(x\) and \(y\) individually profit more from, than their current match.

<table>
<thead>
<tr>
<th>Preference list for different mentor matching criteria:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Pareto_0:</strong></td>
</tr>
<tr>
<td><strong>Pareto_1:</strong></td>
</tr>
<tr>
<td><strong>Pareto_2:</strong></td>
</tr>
<tr>
<td><strong>Pareto_3:</strong></td>
</tr>
<tr>
<td><strong>Pareto_4:</strong></td>
</tr>
</tbody>
</table>
3.4.2.1 Handling Unequal Mentor-Mentee Sets

In subsection 3.4.1, we have explained how feasible candidates are retrieved for help-requesters and probable mentors. We also describe the methods used to sort these candidate lists according to the matching criteria specified by candidates. By applying the methods described previously, we generate 2 preference lists: A help-request object to list of mentors list and a mentor to list of help-request objects (mentee) list. An important point to be noted is that the size of these lists are most likely unequal i.e. the number of available users who can be probable mentors, will mostly be different from the number of help-requesters for that matching cycle (probable mentees). We cannot directly apply the Gale-Shapley algorithm for finding stable matches since it is applicable for equal size sets of elements. We make both preference lists to be of equal size by processing these lists as mentioned below.

1. Adding infeasible candidates last:
   The preference list for a help-request object (pertaining to a mentee) may not contain all the possible available users who are probable mentors. The reason for this is that there could exist many users who are available, but either have not registered for the course corresponding to that help-request object, or whose price is higher than the mentee’s budget. These candidates are infeasible for the given mentee, but without including them, our preference lists would be incomplete. In order to avoid this, all infeasible candidates are appended to the end of each user’s preference list. An analogous method is followed in the case of mentors and their infeasible mentee matches. By adding these candidates to the end of each user’s preference list, we ensure that they are given a low priority when running the Gale-Shapley algorithm.

2. Handling unequal sets:
   Even after ensuring that all probable mentors are in all the mentees’ preference lists and vice versa, we still face the problem of unequal sets. In most scenarios, the number of help-requests submitted (and thus the number of help-request objects) would not be the same as the number of available candidate mentors.
For example, there could be 10 available candidate mentors and just 7 help-
requests for a given matching cycle. In such a scenario, we must add dummy
nodes to equalize both sets. We do this by randomly picking a key from the
smaller set (the mentee set in our example). We use the reverse of the preference
list of this randomly chosen user (a random mentee in our example) as our
dummy preference list. We reverse the chosen preference list since we do not
want an actual user to get a lower preference due to a dummy node. Now, we
must add dummy nodes (3 in our example) to the smaller set such that these
dummy nodes’ preference lists contain all the users of the other set. Finally,
for each dummy node that we added in the smaller set, we append a random
permutation of these dummy nodes to each user’s preference list who belongs
to the larger set (the mentor set in our example).

3.4.2.2 Gale-Shapley Algorithm

Once both mentor and mentee preference lists are of equal size, we apply the Gale-
Shapley algorithm to find stable matches. Algorithm 4 shows the Gale-Shapley
algorithm for the case of Quet’s mentor-mentee matching.
Algorithm 4: The Gale-Shapley algorithm applied to Quet to find stable mentor-mentee matches.

1: function GETMENTORMENTEEMATCHES(M, P) \hspace{1em} \triangleright \text{“M”: Dictionary of mentor preference lists, “P”: Dictionary of mentee preference lists.}
2: \hspace{1em} S \leftarrow \text{[ ]} \hspace{1em} \triangleright \text{Store stable matches in } S.
3: \hspace{1em} m \leftarrow \text{free} \hspace{1em} \forall m \in M \hspace{1em} \triangleright \text{Make all mentors free.}
4: \hspace{1em} p \leftarrow \text{free} \hspace{1em} \forall p \in P \hspace{1em} \triangleright \text{Make all mentees free.}
5: \hspace{1em} \text{while } \exists \text{ free mentor } m \text{ not yet matched with a mentee } p \text{ do}
6: \hspace{1em} \hspace{1em} p \leftarrow M[m].next() \hspace{1em} \triangleright \text{Get } m \text{'s most preferred mentee, that } m \text{ hasn’t tried matching with yet.}
7: \hspace{1em} \hspace{1em} \text{if } p \text{ is free then}
8: \hspace{1em} \hspace{1em} \hspace{1em} S \leftarrow S + [(m, p)] \hspace{1em} \triangleright \text{Tentatively store this match.}
9: \hspace{1em} \hspace{1em} \text{else if } (m', p) \in S \text{ then} \hspace{1em} \triangleright \text{Another tentative match of } p \text{ exists.}
10: \hspace{1em} \hspace{1em} \hspace{1em} \text{if } p \text{ prefers } m \text{ to } m' \text{ then}
11: \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} S \leftarrow S - [(m', p)] \hspace{1em} \triangleright \text{Remove previous match.}
12: \hspace{1em} \hspace{1em} \hspace{1em} \text{\hspace{1em} } \hspace{1em} \triangleright \text{ } m' \text{ becomes free.}
13: \hspace{1em} \hspace{1em} \hspace{1em} \text{else}
14: \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} S \leftarrow S + [(m, p)] \hspace{1em} \triangleright \text{Tentatively store this match.}
15: \hspace{1em} \hspace{1em} \hspace{1em} \text{else}
16: \hspace{1em} \hspace{1em} \hspace{1em} \hspace{1em} \text{end if}
17: \hspace{1em} \hspace{1em} \text{end if}
18: \hspace{1em} \text{end while}
19: \text{return } S

The Gale-Shapley algorithm for Quet’s case of matching mentors to mentees works by initializing all mentors and mentees to be free. Each mentor \( m \) tries to match with his or her most preferred mentee \( p \). Every mentee chooses to match with that mentor who tries to match with him or her, and who the mentee prefers over all other mentors who have tried matching with him or her. If \( p \) is currently unmatched, we tentatively add the match \((m, p)\) to the matched set \( S \). If the mentor \( m \) tries to match with a mentee \( p \) who is already tentatively matched up with another mentor \( m' \), we either break the match or keep it. If \( p \) prefers \( m \) over the previously matched mentor...
we remove the match \((m', p)\) from \(S\) and add the tentative match \((m, p)\) to \(S\).

Each mentor who remains *free* or unmatched after the previous step subsequently tries to match with his or her next preferred mentee (according to his or her preference list). We continue this as long as there are unmatched mentors and mentees. Finally, we reach a state wherein all the mentors have been matched with mentees and all matches made are stable. Harrenstein et al. [21] provide many more such interesting applications and explanations of the Gale-Shapley algorithm in various fields, including hospital-patient matching, kidney-donor matching, supply-demand matching in economics, etc.

An important point to remember is to prune infeasible and dummy matches from the stable matches obtained. In order to do this, we must also maintain the set of feasible candidates for each user. Once the stable matches have been found by applying the Gale-Shapley algorithm, we must only retain those matches that are between candidates that are feasible for one another.

Thus, the final output is a list of pairs for mentor to (mentee, help-request ID) matches. Once we obtain these pairs, we notify matched mentors as explained in section 3.2.

### 3.5 System Architecture and Design

In this section, we provide a brief overview of the system architecture of Quet. We also describe the design goals that had been kept in mind when implementing Quet.

#### 3.5.1 Backend Components

The Quet backend system runs entirely on DigitalOcean [12]. Different backend components were programmed in Python 2.7.6 [36]. Quet runs on a Python Flask [37] server and uses SQL (MySQL) [29], NoSQL (MongoDB) [28] and GraphDB (Neo4j) [30] data stores. Quet uses Celery [6] and celery beat [7] with a RabbitMQ [38] message broker to handle asynchronous and periodic tasks respectively. The
targeted clients for Quet are smartphone users running any mobile OS. Quet has been built as a platform-agnostic hybrid mobile application. Our current deployment of Quet is for Android users. The system is flexible enough to incorporate a Website-based application at a later point in time. This will enable web browser support as well. In this section we describe various backend components used in Quet.

- **DigitalOcean server details:**
  DigitalOcean is a pay-per demand cloud infrastructure provider. We use a DigitalOcean virtual machine as our compute engine and cloud computing platform. Currently, Quet is running on a dual-core (1.80 GHz) 64-bit Ubuntu 14.04.4 virtual machine with 2 GB RAM and 20 GB storage capacity. Our core business logic (including the application server, database services such as MySQL, MongoDB and Neo4j, Celery tasks, etc.) runs entirely on this DigitalOcean server.

- **Application server:**
  Quet’s application server is written using Python Flask 0.10.1. Flask is a micro web framework that is written in Python. Our decision to choose Flask as the application server was based on numerous advantages offered by the Flask framework. Flask provides support for RESTful request dispatching and is completely compliant with WSGI 1.0. Flask supports many other frameworks for form validation, security, session management, templating, etc. In addition to these advantages, Flask also has support for SQLAlchemy [41], which is a robust Object Relational Mapper for Python and MySQL. These advantages made Flask a clear choice for the application server implementation.

- **Handling asynchronous and periodic tasks:**
  Quet uses Celery 3.1.23 to handle asynchronous tasks and celery beat to handle periodic tasks. Many functionalities within the application should be non-blocking and handled asynchronously in the background, instead of blocking the user. Celery provides us with the ability to serve such asynchronous tasks flexibly and reliably in a distributed manner. In addition to this, Celery also
provides us with tools to monitor the asynchronous tasks being handled. Task queues are used to distribute work across threads and Celery worker processes continuously monitor task queues for tasks to be executed. Celery communicates via messages using a message broker to mediate between clients and workers.

Quet uses Celery and celery beat to handle tasks such as retrieving top-rated mentors, running the matching algorithm periodically according to the matching cycle, sending confirmation emails and other time out notifications to users. We use RabbitMQ 3.2.4 as the message broker for the Celery workers, since RabbitMQ is feature complete and robust.

- **Workhorse:**
  We use Python 2.7.6 to implement all of our business logic. The matching algorithm, various backend request endpoints and Celery tasks are all written in Python. Python was chosen for its ease of integrating machine learning and statistical packages, as well as its high compatibility with other packages and modules, such as those for authentication, templating, routing, etc.

- **Data stores used:**
  We use MySQL 14.14 as the SQL database, MongoDB 3.2.6 as the NoSQL data store and Neo4j 3.0.1 as the graph DBMS within Quet. MySQL is used to maintain user credentials for logging in, as well as their GCM [20] token for sending push notifications. As mentioned previously, we use SQLAlchemy as the ORM for interacting with MySQL from Python.
  MongoDB is used to store user profile information, which includes fields like their courses, mentor ratings list, mentee ratings list, status, profile picture (stored as a base64-encoded string), their budget, price, etc. We use PyMongo 3.2.2 [35] as the MongoDB driver from our Python codebase. A MongoDB collection is also used to queue help-requests submitted by various users for a given matching cycle. We choose MongoDB as our preferred NoSQL data store since it is designed for efficient working in applications that are write-heavy. MongoDB also supports high availability and efficient atomic querying of data.
Additionally, the schema-less flexibility provided by NoSQL and the ability to represent data in a JSON-form makes MongoDB a great option. MongoDB also supports the GeoJSON [19] format and querying based on location-based data which is useful for matching users based on their GPS locations (to be incorporated as part of future work).

Neo4j is used to capture interactions between users who have successfully undergone help-request sessions with each other. In other words, the MongoDB help-requests collection which is periodically dropped as help-requests are consumed in each matching cycle, is translated to a social network in Neo4j. Note that we use the Neo4j GraphDB to capture only those help-request sessions which have been completed between 2 users (not cancelled, or declined). We use py2neo 2.0.8 [34] to interact with our Neo4j graph database from Python. Figure 3.25 indicates the kind of social network obtained. The nodes of this graph represent different Quet users who have participated in help-request sessions. This graph has Helped relationships (edges) between each pair of users who have ever been paired up and completed a help-request session with each other. For example, a typical Helped relationship between 2 user nodes $a$ and $b$ would look as follows:

$$a \xrightarrow{HELPED} b$$

This edge would represent a help-request session in which user $a$ was matched as a mentor to user $b$. The Helped relationship contains many properties to capture information about the help-request session such as the question asked, the course, the ratings given by users to each other, the payment involved, etc. Figure 3.26 shows an example of a typical Helped relationship with its different property values. By capturing these interactions in the form of a social network using Neo4j, we are able to run efficient queries that take into account user’s ego networks. This has also been designed keeping in mind future extensions to Quet in which user preference lists could be personalized, taking into account each user’s social network and the peers they interact the most with.
Figure 3.25: Example of the social network obtained by modeling help-request sessions between users in Neo4j.

3.5.2 Frontend Components and Other Functionalities

Quet has been designed so that it can be used for any mobile platform. The current deployment of the application is for Android, but this can be easily extended for other mobile platforms. This is achievable through the use of a hybrid mobile application framework like React Native [40]. In this section, we describe external software/modules used in Quet to provide extended functionality.

- **React Native frontend:**
  React Native allows us to build native mobile applications using just JavaScript. The mobile application interface that is developed using React Native is indistinguishable from native mobile applications that would have been built using Android or Objective-C/Swift (for an iOS app). Furthermore, React Native is more native than other similar frameworks like Ionic [23], which is based on Apache Cordova [2]. Though Cordova-based frameworks like Ionic allow developers to make native API calls, the bulk of the application is HTML and
Figure 3.26: Example of a Helped relationship wherein user 2 has helped user 1.

JavaScript inside a WebView, thus only being able to approximate the look and feel of a native application. React Native on the other hand, is completely integrated with the native mobile OS, making it feel exactly like a native application. The added advantage of a native application is performance and reduced response latency. These factors are extremely important when considering user experience.

- **Push notifications:**
  We make use of Google Cloud Messaging to handle push notifications and the in-app chat. GCM is very simple to use, reliable and has support for both Android as well as iOS. In order to integrate GCM with our application, we have made use of pushjack 1.2.1 [33]. Pushjack provides support for push notifications using APNS (Apple Push Notification Service) [3] for iOS, as well as GCM for Android.
• **User authentication:**
  Quet authenticates users by sending a confirmation email using Flask-Mail [16]. Flask-Mail is a Flask extension that provides a simple interface to set up SMTP with a standalone Flask application. The confirmation email is send asynchronously using Celery. Token generation is done securely using the itsdangerous [24] package. User authenticity is checked for all routes requiring login by validating authorization headers by encoding and decoding JSON Web Tokens using the jwt [25] package.

• **Overall working:**
  Figure 3.27 goes over the entire system architecture of Quet. The frontend and backend communicate using REST APIs defined as endpoints in Quet’s Flask application server.

### 3.5.3 Design Goals

At a high level, we want our system to cater to the following design goals:

• **Low latency and response time:**
  With huge volumes of data and multiple simultaneous requests, there is not only a need for constant data availability, but also for minimum data retrieval delays. Thus, latency has emerged to be one of the most important parameters in order to judge the performance of a distributed systems application like Quet. In applications like Quet which are interactive and involve complex matching algorithms, retrieval of large amounts of data and transferring large amounts of data over the network, reducing latency is an important requirement. Designing applications that allow faster data retrieval also improve user experience by reducing the application’s response time. Our design decisions have been made taking these factors into consideration. MongoDB, Neo4j and their respective interfaces with the underlying Python backend server are designed for write-heavy applications and provide data at low latency. As men-
Figure 3.27: Quet’s system architecture diagram.

As mentioned in the previous section, the use of React Native also reduces application response time as compared to similar frameworks based on WebViews.
• **Fault tolerance and system availability:**
  Fault tolerance is a must since we do not want data loss and the system must be available to serve client requests at all times. We run the matching algorithm periodically according to the matching cycle. For help-requests to be served without loss, we must ensure fault tolerance. MongoDB implements replica sets for fault tolerance and thus is a great choice for achieving this.

• **Finding the best possible matches:**
  By taking into consideration the matching criteria of both, the matched mentor as well as the mentee, Quet ensures that stable matches can be found. The matching algorithm preference list generation also indirectly prefers users with high ratings and also users who are altruistic while offering help. The Gale-Shapley algorithm used for finding stable matches is very well known and has been applied in numerous fields for finding stable matches.

• **User privacy and security:**
  The privacy and security of users is of utmost importance in order to maintain a strong user-base. Users who wish to use Quet must consent to use the application first. This requires them to go over a series of consent documents as explained in section 3.1. Since Quet has been designed to be deployed only at the UIUC campus (for now), we ensure that users consenting to use the application are valid UIUC students by sending them a confirmation email on their Illinois email ID. Furthermore, passwords are stored as an MD5 hash in the backend and the hashing is done at the client-end to avoid security issues due to Man-in-the-middle attacks. Each endpoint requiring users to be logged in, checks for the presence of a valid authorization header to proceed. Furthermore, when a requesting user requests to read or update information about his or her matched user, the backend ensures that the user has permission to do this by employing a 2-way check for both users’ status values. Thus, Quet has been designed to keep in mind user privacy and security.
Ensuring data integrity:
Data that is stored in the backend data stores must be consistent and accurate. Data corruption should be avoided at all costs. MongoDB and Neo4j are consistent by default. MySQL is obviously ACID consistent and thus retrieving the latest update of data is not an issue. Furthermore, SQL injection-like attacks are impossible due to the use of an ORM and sufficient precautions when parsing form inputs. Thus, our design choices in Quet ensure data integrity.

Scalability:
We foresee that the system would have users from various regions of the world. In order to sustain such a large user base and diverse geographies, the system should be designed to scale horizontally. With the addition of other frontends such as web browsers, the need for scalability will grow even further. All design decisions made while implementing Quet have kept scalability as the top priority. The DigitalOcean servers used can be scaled easily to provision a larger user base, increased storage and more compute power. MongoDB is well-suited for scaling and can provide nearly 100% availability even for millions of operations per second on nearly 100 billion documents. Neo4j delivers extremely fast read and write performance, while still protecting data integrity. It combines native graph storage, scalable architecture optimized for speed, and ACID compliance to ensure predictability of relationship-based queries. Neo4j has inbuilt query caching [31] that enables faster response times and efficient retrieval of data. It is easy to learn and use and can be easily integrated with a wide array of programming languages. The Cypher Query Language [10] is very intuitive and query results are easy to visualize using the native GUI provided by Neo4j. Moreover, Neo4j has extensive support for graph based machine learning algorithms such as PageRank, HITS, etc. Using Neo4j to capture interactions between users can be very useful for future extensions of Quet which make use of social network analysis and personalized preferences for improving matches. Celery and RabbitMQ are both designed for distributed asynchronous task processing and thus will aid in scaling Quet.
• **Power efficiency:**
A lot of applications including Quet, run as background processes and share the same common battery on the mobile phone. Therefore, designing power efficient applications is an important responsibility. Quet aims at ensuring power efficiency in spite of having to periodically send user’s notifications. Quet avoids excess battery consumption by avoiding unnecessary communication with the server and by replacing polling requests with asynchronous background Celery tasks which offload the polling task from the mobile device to the server.

• **User experience:**
Apart from the above design goals which are important for providing a holistic user experience, explicit measures must be taken to ensure that the application is easy to use, intuitive, abstracts out the underlying complexities and is aesthetically pleasing. The application should provide for a smooth user experience and should not crash even on erroneous user inputs. Quet has been designed keeping in mind simple design principles so that it resembles a native mobile application as much as possible, in spite of being created using React Native. Descriptions for various fields are short and easy to understand. Buttons are not hidden in any way and very easy to find. It is also fairly simple to navigate between different pages on the application.

• **Data visualization:**
A picture is worth a thousand words. Data visualization techniques greatly aid in analyzing trends, patterns and various other characteristics in large amounts of complex data. Figures 3.25 and 3.26 show us examples of the kind of visualization possible with Neo4j. Showing an ego network view of this to users can also be done in order to give them an idea of their interactions as compared to the interactions of their past matches.

Thus, we can see that Quet has been designed keeping in mind multiple design goals. All of the components used within Quet have been chosen carefully in order to comply with these design goals and to aid future extensions to Quet.
In this chapter, we describe the proposed experimental setup for evaluating the usefulness of Quet. We also analyze various performance metrics (like latency, application response time, etc.), which affect the ease of use of the application. By evaluating Quet on these metrics, we are able to gauge how inclined people would be to use such an application.

4.1 Survey Questions

In this section, we describe the different online survey questions that we had designed in order to understand how applications like Quet can help students. These questions have also been aimed at understanding the level to which students currently collaborate with their peers. The proposed method of survey evaluation was to divide the total number of participants equally into a control group that does not use the application, and a test group that does. In order to remove the effects of selection bias, we planned to ask a series of online pre-experiment questions to participants belonging to both groups. It was then planned to allow test group participants to use Quet for a time period of around 10 days. At the end of this 10-day period, test group participants would be paid a dollar equivalent of the Quet Credits that they have earned on the application. After the 10-day period, we had planned on asking participants belonging to both groups to fill out the same set of online survey questions. Unfortunately, we were unable to conduct this experiment due to a lack of participants, but informal feedback from students conveys that Quet could definitely
be useful. The questions that were designed as part of the pre and post-experimental online surveys have been listed below.

1. **How well do you think you are doing in class?**
   Options ranged from 1 (Not good) to 5 (Excellent).

2. **How many of your peers do you interact with on an average, per week?**
   Options were: None, 1-2, 3-4, 5-6, 7+.

3. **Getting help from my peers is useful to me.**
   Options were: Strongly Disagree, Disagree, Undecided, Agree, Strongly Agree.

4. **TA office hours or interaction sessions with instructors are frequent enough to get adequate help with questions I encounter in the course material.**
   Options were: Strongly Disagree, Disagree, Undecided, Agree, Strongly Agree.

5. **Piazza and other online forums are adequate to receive help with questions I encounter in the course material.** Options were: Strongly Disagree, Disagree, Undecided, Agree, Strongly Agree.

6. **I enjoy helping my peers.** Options were: Strongly Disagree, Disagree, Undecided, Agree, Strongly Agree.

7. **I find myself stuck on questions which can be easily resolved by speaking to peers in my class.** Options were: Strongly Disagree, Disagree, Undecided, Agree, Strongly Agree.

8. **I am comfortable asking questions in front of everyone in class.** Options were: Strongly Disagree, Disagree, Undecided, Agree, Strongly Agree.

9. **I generally get answers to questions I have in class by:** Options were: Asking the instructor/TA, Asking my peers, Searching over the internet, Using Piazza/Other applications.
10. I believe that teaching is the best way to learn something new.
Options were: Strongly Disagree, Disagree, Undecided, Agree, Strongly Agree.

11. I do not mind spending a small amount of money (cost of coffee) to receive help for questions I encounter in my courses.
Options were: Strongly Disagree, Disagree, Undecided, Agree, Strongly Agree.

4.2 Performance Metrics

As per our design goals in subsection 3.5.3, latency is one of the most important performance parameters of today’s interactive applications. Keeping this in mind, we have thoroughly evaluated various components of our system in terms of latency. Latency can help us understand application response time for various operations, through which we can then gauge the ease of using Quet. This becomes increasingly important when estimating the extent to which Quet will be popular amongst users.

4.2.1 MongoDB Performance Metrics

MongoDB collections are used to store user profile information. We also make use of a MongoDB collection to temporarily store the help-requests that have been submitted in a particular matching cycle. Common operations like showing the user his or her profile, editing the user's profile, etc. require reads and writes to MongoDB.

4.2.1.1 Read Latency

Showing a user his or her profile information requires the backend to query the MongoDB profile collection and send this data to the user via a REST response. This profile information includes the user’s display name, available Quet Credits, profile picture, courses signed up for, budget, cost and mentor matching criteria. Figure 4.1 shows the CDF of the amount of time required to retrieve a user’s profile information from the MongoDB profile collection, without taking into account network latency.
We can see that this read operation is extremely fast and in around 80% of the cases, this operation only requires around 2.7 milliseconds.

Figure 4.1: Profile load times without taking into account network latency.

Figure 4.2 shows the CDF of the amount of time to retrieve a user’s profile information from the MongoDB profile collection and also transfer this to the user via the network. Note that sending the user’s profile information requires sending his or her profile picture over a network connection, due to which this is a lot of data. Even when taking into account network latency, this operation is fast and can be completed in around 0.21 seconds about 80% of the times.

4.2.1.2 Write Latency

When a user edits his or her profile, an update or write is made in the document corresponding to that user in the MongoDB profile collection. The user may edit various aspects of his or her profile including the profile picture, course list, budget,
cost and mentor matching criteria. As expected, an edit profile request requiring the profile picture to be updated is heavier and requires more time than an edit profile request that only updates the other fields. Figure 4.3 shows us the CDF of the amount of time required to make a simple edit profile request (without changing the profile picture), without taking into account network latency. We can see that this write/update operation is also extremely fast and in around 80% of the cases, this operation only requires around 2.5 milliseconds.

Figure 4.4 shows the CDF of the amount of time required to make these simple profile updates, taking into account network latency. We can see that despite network latency, this write/update operation is still very fast and can be completed in around 0.107 seconds in about 80% of the cases.

If we were also to update the profile picture, this operation becomes heavier as we can see in figure 4.5. This figure shows us the CDF of the amount of time required to make an edit profile request which also updates the user’s profile picture,
without taking into account network latency. In spite of the update being heavier, this operation completes in about 51 milliseconds in about 80% of the cases.

Figure 4.6 shows us the CDF of the amount of time required to make a heavy (by updating the profile picture as well), edit profile request which also takes into account the network latency. Note that this request requires sending the updated profile picture via a REST request to the backend server. We see that this operation can still be completed in around 153 milliseconds in about 80% of the cases which is still very fast and does not hinder user experience at all.

4.2.2 Neo4j Performance Metrics

Neo4j is used to maintain a graph database to store help-request session information in the form of a social network. Every time a mentor and mentee complete a help-request session, nodes corresponding to these users are created in the graph
Figure 4.4: Simple profile edit times taking into account network latency.

database (if not already present) and a directed “Helped” relationship is added from the mentor node to the mentee node. This “Helped” relationship stores information pertaining to the help-request like the session number between these 2 users, the estimated time entered when requesting help, the mentee and mentor matching criteria, the actual question, the course it pertains to, location of the session, payment amount and the ratings given by each user to the other.

4.2.2.1 Read Latency

Each help-request session between a mentor and a mentee is denoted by a session number, which conveys the number of times that the mentor has helped this mentee before. In order to enter the correct session number for the “Helped” relationship to be added between the mentor and mentee nodes, we must query the Neo4j graph database for all “Helped” relationships between the particular mentor and mentee.
Figure 4.5: Heavy profile edit times without taking into account network latency.

Figure 4.7 shows the CDF of the amount of time required for a typical match query to be run. We can see that in around 80% of the cases, this operation can be completed in about 60 milliseconds. Also note that the time calculation for this operation takes into account network latency as well.

4.2.2.2 Write Latency

As mentioned in subsection 4.2.2, we create nodes and relationships for users corresponding to help-request sessions completed between these users. Note that in order to achieve this, we must first check for the existence of a node corresponding to each particular user (to avoid duplicate node creation). Thus, this write operation requires us to use the merge operation to ensure that a pattern exists in the graph by creating it, if it does not exist already. After this step, we then create the “Helped” relationship between the mentor and mentee nodes. Figure 4.8 shows us the CDF of
the amount of time required to carry out this write operation. We can see that in 80% of the cases, this operation can be completed in about 0.23 seconds. Note that even for this operation, the time calculation takes into account network latency.
Figure 4.7: Time taken to run a typical match query in Neo4j to find nodes and a particular “Helped” session between these nodes.
Figure 4.8: Time taken to create (if not present) nodes for the mentor and mentee and then create a “Helped” relationship containing information mentioned in subsection 4.2.2, amongst these nodes.
CHAPTER 5

CONCLUSION

In this thesis, we propose a novel application of the sharing economy-based model in the context of knowledge sharing. We implement this idea in the form of a mobile application for peer-to-peer help and knowledge sharing, called Quet. Quet uses a combination of social reputation-based incentives as well as hard incentives (Quet Credits translated to a dollar equivalent). To the best of our knowledge, we are the first to explore the use of a sharing economy-based model for on-demand tutoring. Informal feedback from plausible users indicates that Quet can definitely help students and help make problem solving efficient and less time-consuming for them. Chapter 4 also evaluates the ease of use of Quet in terms of low latency and application response time. Our results from chapter 4 confirm that we have been able to achieve our design goals mentioned in subsection 3.5.3. When comparing Quet with other similar applications for finding on-demand tutors, we observe the following advantages of Quet over these services:

- Other services mostly involve matching students with actual tutors, so there is no peer-to-peer help. Students and tutors are considered as two mutually exclusive groups. Quet overcomes this by allowing students to take on the role of mentors as well as mentees.

- To the best of our knowledge, most of these apps show students all available tutors and students are responsible for selecting which tutor they wish to be matched with. Quet offloads this responsibility from the students and does the matching on its own. Thus, students do not have this extra overhead and are simply responsible for submitting their help-requests. Quet takes care of
• The costs that students incur when matched with help is much higher in the case of these other applications. The reason for this is that students are matched with actual tutors charging around $75 per hour, or so. Quet is designed for more informal help, which does not necessarily require students to be matched with a professional tutor. Due to this, the charges for students are akin to the cost of a cup of coffee, making help much more affordable.

• Competitor applications do not focus on peer-to-peer learning and help. Quet is more useful for promoting in-class collaboration amongst peers.

• The model of Quet is currently limited to the classroom, however, peer-to-peer help can be extended to many domains.

5.1 Future Work

Unfortunately, due to lack of participation, we were unable to run the experiment described in chapter 4. As part of future work, we hope to run this experiment and evaluate the differences in answers obtained to the online survey questions, to understand the advantages obtained by the introduction of Quet. The version of the application that has been currently developed has a basic easy-to-use UI. We wish to improve the UI to make the application friendlier. Due to issues in iOS licensing, we have only deployed the Android version of Quet. As part of future work, we wish to resolve these licensing issues and also develop a version compatible with iOS devices. Quet has been designed for scalability and the design has been made keeping in mind the addition of machine learning algorithms and an improved matching mechanism. We wish to personalize the user preference list generation by making use of historical data that has been mined from past interactions of users. Fortunately, due to the use of Neo4j to represent help-request interactions between users in the form of a social network, this type of social network analysis and machine learning can be easily integrated without changing the application’s architecture and
design. Finally, we wish to deploy Quet at large courses within UIUC and evaluate the user response. As mentioned earlier, we also wish to extend Quet for on-demand peer-to-peer knowledge sharing in general, not just limited to the classroom context.
REFERENCES


[34] py2neo, 2016. URL http://py2neo.org/2.0/.


97


