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

Martijn de Vos
Delft University of Technology

Georgy Ishmaev
Delft University of Technology

Johan Pouwelse
Delft University of Technology

Abstract

Matchmaking is a core enabling element in peer-to-peer markets. To date, matchmaking is predominantly performed by proprietary algorithms, fully controlled by market operators. This raises fairness concerns as market operators effectively can hide, prioritize, or delay the orders of specific users. Blockchain technology has been proposed as an alternative for fair matchmaking without a trusted operator but is still vulnerable to specific fairness attacks.

We present MATCH, a decentralized middleware for fair matchmaking in peer-to-peer markets. By decoupling the dissemination of potential matches from the negotiation of trade agreements, MATCH empowers end-users to make their own educated decisions and to engage in direct negotiations with trade partners. This approach makes MATCH highly resilient against malicious matchmakers that deviate from a specific matching policy. We implement MATCH and evaluate our middleware using real-world ride-hailing and asset trading workloads. It is demonstrated that MATCH maintains high matching quality, even when 75% of all matchmakers is malicious. We also show that the bandwidth usage and order fulfil latency of MATCH is orders of magnitude lower compared to matchmaking on an Ethereum blockchain.

CCS Concepts • **Computer systems organization** → **Peer-to-peer architectures**; • **Networks** → *Peer-to-peer protocols.*

Keywords Fair Matchmaking, Decentralized Middleware, Matching Middleware, Peer-to-Peer Markets

ACM Reference Format:
Martijn de Vos, Georgy Ishmaev, and Johan Pouwelse. 2020. MATCH: A Decentralized Middleware for Fair Matchmaking In Peer-to-Peer Markets. In *21st International Middleware Conference (Middleware '20), December 7–11, 2020, Delft, Netherlands.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3423211.3425678

Middleware '20, December 7–11, 2020, Delft, Netherlands
© 2020 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8153-6/20/12.
https://doi.org/10.1145/3423211.3425678

1 Introduction

The deployment of peer-to-peer markets by companies operating in the sharing economy has been hailed to boost the global economy [34]. Beyond the promises of increased economic welfare, the broader appeal of the sharing economy also lies with the development of new modes for the sharing of unused or underutilized assets, such as cars and houses. Estimations on the impact of the sharing economy suggest an increase in global revenue from $14 billion in 2014 to $335 billion by 2025, partially enabled by major platforms such as Uber (ride-hailing) and AirBnb (house-sharing) [42].

The effect of these platforms on peer-to-peer markets, however, is not unequivocal. It has been argued that market operators exploit their prominent position and charge high transaction fees for their role as intermediary [44]. Market operators gain unprecedented power through the control of all the key enabling elements for electronic marketplaces, including settlement, arbitrage, and matchmaking [3, 14, 40]. The latter element is of particular interest as at the dawn of e-commerce matchmaking solutions were envisioned to create open, fair, and competitive markets on the Internet [52].

Matchmaking in electronic markets can be considered as the process of mediating between supply and demand [54]. Currently, centralized matchmaking is the approach taken by most commercial market operators [3, 40]. With centralized matchmaking, market operators deploy proprietary servers that are optimized to efficiently match new buy and sell orders with existing ones within a specific domain. A key advantage of centralized matchmaking is that the market operator can match incoming orders with the (current) best compatible order since they maintain all market information.

Unfortunately, this also enables market operators to exploit the marketplace through the practices of unfair matchmaking to increase intermediary revenues [27]. Manipulation in the matchmaking process was exposed through analysis of different e-commerce platforms and financial exchanges [35, 40]. An emblematic example of this issue is the practices of Uber, implementing price discrimination and phantom matches to manipulate the behaviour of users [9]. It has recently been demonstrated through experiments that the matchmaking algorithm of Uber undermines revenues of drivers to the advantages of the platform operator [7]. As researchers point out, unfair matchmaking is a complex, multilayered issue that can not be mitigated only with algorithmic adjustments [7]. We suggest that this problem requires a next step towards a different approach to matchmaking in peer-to-peer markets.

It is technologically feasible to have market participants carry out the matchmaking process themselves, without trusted operator. In particular, blockchain technology provides the means to record and match market orders on a distributed ledger [50]. Smart contracts, self-executing programs stored on a blockchain, are capable of executing the matchmaking logic [33]. Even though it seems like an appealing solution to fairness issues, the consensus algorithm managing the blockchain ledger is vulnerable to various attacks against fairness, specifically, *transaction re-ordering* and *front-running* [21, 29, 30]. These attacks effectively allow consensus participants to influence how specific orders are matched. In addition to these threats, scalability issues inherent to all the blockchain protocols based on a global consensus carry significant limitations on the speed of matchmaking, as we will experimentally show in Section 6.3 [16].

The ineffectiveness of matchmaking on a blockchain is also identified by various decentralized exchanges that are operated by a blockchain, also called DEXes [21, 31, 57]. In response, these DEXes opt for a federated approach where any participant can host a matchmaking server and can act as a matchmaker. In practice, most orders in these DEXes are managed by servers that are under the control of the exchange operator and therefore still carry limitations on the achievable level of fairness desirable on such markets.

In summary, there is a dilemma of choice between two desirable properties of matchmaking mechanisms. *Efficiency* of matchmaking achievable through the concentration of orders by a trusted central party, versus provable *fairness* of matchmaking achievable with the transparency of decentralized on-chain matchmaking. We argue, however, that this dilemma does not present an insurmountable obstacle for the implementation of efficient and fair matchmaking solutions.

Contributions. We present MATCH, a decentralized middleware for fair matchmaking in peer-to-peer markets. Our solution is based on the principle of strictly decoupling the matching process from the negotiation of trade agreements. Our first contribution is the MATCH protocol (Section 4) where any user can act as a matchmaker for others. Matchmakers store open orders created by users, match incoming orders with existing ones, and inform order creators about potential matches. Users then engage in trade negotiation with prospective counterparties. This approach makes MATCH highly resistant against matchmakers which deviate from a standard matching algorithm. The second contribution is the generic MATCH middleware architecture (Section 5). MATCH does not rely on the specifications of orders, and is therefore re-usable across different trading domains.

We devise the first decentralized and fair alternative to the Uber ride-hailing market (Section 6.1), to the best knowledge of the authors. Even when 75% of all drivers prioritize their own ride services during matchmaking, negotiated matches in our market maintain a high quality. Furthermore, with a real-world asset trading workload (Section 6.2) we show that MATCH is asset-agnostic, enabling the deployment of open and universal matchmaking infrastructures. Finally, we show that MATCH has bandwidth usage and order fulfil latencies that are several orders of magnitude lower compared to matchmaking on an Ethereum blockchain (Section 6.3).

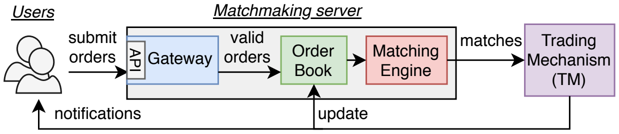


Figure 1. A generic model for centralized matchmaking in peer-to-peer markets, using a single matchmaking server.

2 Towards Decentralized Matchmaking

In our approach, users in a peer-to-peer market conduct the matchmaking process themselves. To elaborate on the components used in our solution, we first devise a generic, centralized model for matchmaking. We then identify technical concerns that arise when decentralizing this model.

2.1 Centralized Matchmaking

We devise a generic model for centralized matchmaking in peer-to-peer markets, see Figure 1. This model is the starting point for our fair matchmaking solution and is inspired by existing architectures that have been widely used by financial markets [35, 53]. Users create *orders*, which they then submit to the matchmaking server. An order specifies interest to buy or sell assets, resources, or services (orders are further discussed in Section 4.1). Many markets deploy one or more gateways that filter out invalid orders and mitigate targeted attacks on the matchmaking server, such as a DDoS attack [35]. Valid orders are inserted in the *order book*, a local data structure that bundles all open and valid orders.

Upon the insertion of a new order in the order book, it is forwarded to the *matching engine*. This component searches for existing orders in the order book that match with the newly submitted order. In particular, an incoming order should be matched with the current best compatible order(s). Whether two orders match is predicated by a *matching policy*. For example, the price-time strategy is a predominant matching policy in financial markets where orders are first matched based on their price and then on their creation time (prioritizing older orders) [35]. The matching engine can establish multiple matches for a single order, e.g., a buy order for a large number of assets can be matched with multiple (smaller) sell orders. Established matches should be "executed", which is an application-dependent operation. In a financial exchange, for example, the specified assets in the matched orders should be exchanged between the order creators. In a ride-hailing market, however, the driver and passenger should be put in contact with each other. We model the component that executes established matches as the *trading mechanism* (TM), which we consider external to the model in Figure 1. After established matches have been executed by the TM, the affected orders in the order book are updated (or removed if they are completed), and the order creators are notified of the match execution.

2.2 Decentralized Matchmaking

The model in Figure 1 allows the server operator to conduct unfair matchmaking by manipulating the matching engine (or policy) to hide, prioritize, or delay specific orders. To address this situation, we propose a solution where users (order creators) themselves carry out matchmaking while ensuring that no single user can authoritatively decide on how the orders of other users are executed. We first consider a basic, decentralized architecture where every user operates a single matchmaking server. A user then submits a new order to all matchmaking servers, and all servers forward established matches to the same TM. The TM executes incoming matches in a FCFS manner. This approach, however, raises the following technical concerns:

1) How does a decentralized matchmaking architecture process matches of the same order found by distinct matchmaking servers? Deploying a single matchmaking server prevents the situation where distinct matchmaking servers submit the same or different (valid) matches for the same order to the TM. Assuming a FCFS execution of incoming matches by the TM, having multiple matchmaking servers sending matches to the TM likely results in the situation where matches of the same order are executed multiple times, resulting in an incorrect order state. To ensure correctness, our replicated matchmaking architecture requires additional coordination.

One solution involves the periodic execution of a Byzantine fault tolerant consensus protocol by matchmaking servers to decide which matches are sent to the TM. Unfortunately, reaching consensus is a resource-intensive process and existing protocols, e.g., PBFT [12] or Proof-of-Work [25], do not scale when the number of matchmaking servers or orders increases [55]. We avoid the need for consensus by having users *negotiate* trade agreements with potential counterparties (further described in Section 4.4). Upon a positive negotiation outcome, these trade agreements, digitally signed by both parties, are sent to the TM and executed. Matchmakers only *notify* users about potential matches for their orders. Since it is in the best interest of users to correctly manage their orders, rational users will not sign trade agreements that would result in an incorrect order state.

2) Is it required to disseminate a new order to all matchmaking servers? In the architecture described above, a user sends a new order to all matchmaking servers. This results in full replication of the order book, under the assumption that all matchmaking servers eventually receive every new order. The problem is that a flooding-based dissemination strategy leads to severe performance degradation when the number of matchmaking servers increases, as illustrated by deployed peer-to-peer applications like Gnutella [13]. We address this concern by sending a new order to a small, random subset of all matchmaking servers such that it is still likely that at least one honest matchmaking server receives compatible orders (further described in Section 4.2).

3 System and Threat Model

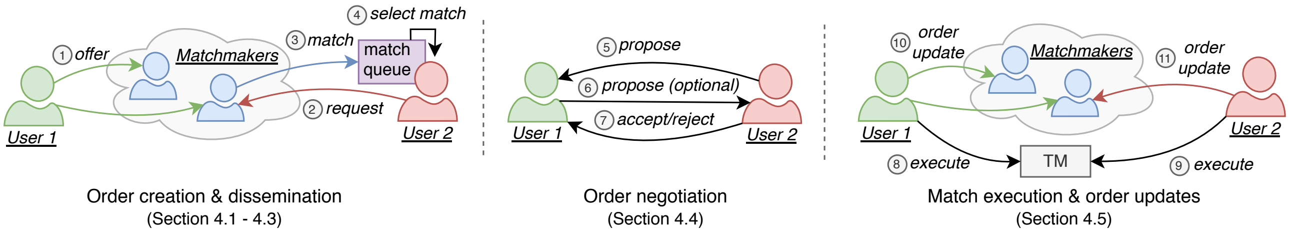
We address the aforementioned concerns and present a decentralized middleware for fair matchmaking, named *MATCH*. We now outline the system and threat model of MATCH.

Market and order model. We adopt a continuous market model where orders are matched in a FCFS manner. This model is commonly used by peer-to-peer markets (e.g., by Uber). To represent a two-sided market with supply and demand, we introduce two order types: *offers* and *requests*. Offers, respectively requests, indicate interest to sell, respectively buy specific assets, services, or resources. To ensure reusability across different markets, matchmakers in MATCH can host multiple order books and manage orders with differing specifications. Each order has a quantity q, which is an integer value indicating the number of assets, services, or resources being offered or requested. The state of an order can be either *open* (when the order has a positive quantity, $q > 0$), *completed* (when all quantity in the order has been traded, $q = 0$), *cancelled* (when the order has been cancelled by its creator) or *expired* (when the timeout of the order has expired). The structure and content of an order are further elaborated in Section 4.1.

Actors. We refer to an entity in the MATCH network as a *node*. A node can act as a *user*, *matchmaker*, or take on both roles. Users disseminate offers and requests to matchmakers. MATCH requires the active participation of users to negotiate with other users and thus requires users to be online for their order to be completed (also see Section 4.4).

Network. We aim for MATCH to be deployed in a WAN environment. We consider an unstructured peer-to-peer network where each node knows the network addresses of active matchmakers. This can be achieved by maintaining a list of matchmakers, e.g., on a website or through a decentralized publishing network like the Kademlia DHT [36]. New matchmakers enrol themselves on this list while matchmakers leaving the network un-enrol from this list. As we show in Section 4.2, MATCH is able to deal with offline matchmakers that are still enrolled on the list. Users periodically download the latest version of this list to keep up with the set of active matchmakers in MATCH.

Each node possesses a public and private key. The public key is used to identify the node in the network whereas the private key is used to sign all outgoing network messages.



Order creation & dissemination
(Section 4.1 - 4.3)

Order negotiation
(Section 4.4)

Match execution & order updates
(Section 4.5)

Figure 2. High-level overview of the MATCH protocol and the message exchange between users and matchmakers.

We assume that the digital identity of each node in MATCH is tied to a real-world identity, preventing uncontrolled identity creation (also known as the Sybil Attack) [18]. This is not an unrealistic requirement since many electronic marketplaces already impose identity verification in order to participate [17]. Identity verification can, for example, be achieved by using the services of a centralized registration authority. We note, however, that a centralized dependency might be undesirable in marketplaces with a decentralized structure. In such marketplaces, we encourage the use of (semi-)decentralised solutions for identity management, like self-sovereign identities [39, 49].

Threat Model. In this work, our threat model orients around *malicious matchmakers* that deviate from a standard matching policy and match incoming orders according to a custom policy. For example, a matchmaker can deliberately match an incoming offer with the second-best request in their order book and match one of their own offers with the hidden request instead. This threat model also captures collusion, the situation where a subset of matchmakers agrees to match orders according to a custom policy to gain economic benefit as a group. Malicious matchmakers are often driven by economic incentives, e.g., when a matchmaker wants to prioritize its own orders or when a group of matchmakers collectively attempt to drive competitors out of business by ignoring their orders. Malicious matchmakers directly affect market fairness since they treat incoming orders unequally. We assume that cryptographic protocols are secure and that the computational power of adversaries is bounded.

4 The MATCH Protocol

We visualize the MATCH protocol and the message exchange between users and matchmakers in Figure 2. The key idea behind our protocol is matchmakers only inform users about matches, and that users negotiate a trade directly with counterparties. First, users send new offers and requests to one or more matchmakers (① + ②). Matchmakers match incoming orders with existing orders in their order books and notify users about potential matches (③). Users aggregate potential matches of a specific order in a match queue. Some period after receiving the first match for a specific order, a user starts to process matches in the associated match queue (④), starting with the best match, and negotiates with the user behind the matched order (⑤ - ⑦). When the negotiating parties reach a trade agreement (in other words, intend to fulfil their orders with each other), they execute the negotiated trade agreement by sending it to the TM (⑧ + ⑨). The negotiating parties then inform the matchmakers about the executed trade so they can update the state of the affected orders accordingly (⑩ + ⑪). The matchmakers are also informed about a negative outcome during the negotiation process. If an order is still open, a user selects the next best item from the associated match queue, if it is non-empty, and initiates the next negotiation process. This repeats until the match queue is empty or the order is completed. The steps in the MATCH protocol are now further explained.

4.1 Order Creation

In MATCH, users create new orders to indicate their willingness to buy or sell resources, services and assets. Listing 1 exemplifies the structure of an order in MATCH that specifies a transportation request in a ride-hailing market. This order, in JSON format, includes the waiting location of the order creator in the `data` field. The content of the `data` field is flexible and depends on the context in which the order is created. It can contain many attributes and constraints that affect how the order is matched. Each order has a `type` field which is a string value indicating the type of the order. The order type is used by matchmakers to apply the right policies for validation and matching, and to store the order

Listing 1. An order in a ride-hailing market (in JSON format).

```
{"timestamp": "2020-02-24T09:09:19+0000",
 "type": "RIDE",
 "timeout": 3600,
 "is_offer": false, // request for transportation
 "public_key": "82ae2f8f0c473cbdf63b920...",
 "signature": "d54af87c8f8e6d917729d14...",
 "identifier": 5,
 "quantity": 1,
 "data": {
   "latitude": "40.712776",
   "longitude": "-74.005974"
} }
```

in the appropriate order book. In Listing 1, the `RIDE` type indicates an order in a ride-hailing market. Similarly, an order with type `EUR/USD` can indicate an order trading Euro for Dollar. The `is_offer` field is a flag that indicates whether the order is an offer or a request. The identifier in an order is an integer value that indicates the position of the order in the sequence of all orders created by that user. Together with the public key of the order creator, it uniquely identifies an order in the network. The digital signature in an order allows matchmakers to verify its authenticity. Inclusion of the `timeout` value prevents orders from being included in order books for an indefinite amount of time. Finally, each order has a quantity, which is an integer value that specifies the amount of assets, services or resources being offered or requested. After creation, a user serializes the order in an `order` message and sends it to matchmakers.

Users can cancel an open order, say O, at any time by sending a `cancel` message with the identifier of O and its public key to matchmakers. A `cancel` message for O should be sent to the same matchmakers as the `order` message that contained the description of O. Therefore, users keep track of the matchmakers to which they have sent an `order` message. Upon reception of a `cancel` message, matchmakers remove the cancelled order from their order book.

4.2 Order Dissemination

In the replicated matchmaking model proposed in Section 2.2, a new order is disseminated to all matchmakers. We now address concern 2 from Section 2.2 and show how to considerably decrease the fanout of `order` messages (i.e., the number of matchmakers that a specific `order` message reaches) while still ensuring a high probability that a new order reaches a matchmaker with the current best matching order in their order book. Specifically, we send a new order to a random subset of all matchmakers. Let R_{req} and R_{off} indicate the sets of matchmakers that receive a specific request and offer, respectively. The probability that at least one matchmaker will receive both a specific offer and request quickly goes to one, even when the order fanout is low compared to the number of matchmakers. This phenomena is also known as the “birthday paradox” and is in practice exploited when computing hash collisions or when detecting a double spend attack in the Bitcoin network [45].

Determining to how many matchmakers a new order is sent, is key. In particular, we are interested in computing the probability that at least one matchmaker receives a matching offer and request. If we consider a network with 1’000 matchmakers where new orders are disseminated to 50 matchmakers, this probability is given by $\frac{1000-50}{1000} \cdot \frac{1000-50-1}{1000} \cdots \frac{1000-50-49}{1000}$. The probability that there is at least one matchmaker amongst all m matchmakers receiving both an offer and a request, with order fanout f, is equal to:

$$P(R_{req} \cap R_{off} \neq \emptyset) = 1 - \prod_{i=0}^{f-1} \left(\frac{m-f-i}{m}\right) \tag{1}$$

The value of $P(R_{req} \cap R_{off} \neq \emptyset)$ quickly increases when f increases. Even if $m = 100'000$ and $f = 500$ (i.e., orders are sent to only 0.5% of all matchmakers), the probability that at least one matchmaker receives both a matching offer and a request, is already 97.7%. In this setting, it reduces the required fanout of an `order` message from 200’000 (when disseminating a new order to all matchmakers) to merely 1’000, thus significantly reducing the network traffic required for order dissemination. Note that the value of m is known to users in MATCH since they possess a list of all matchmakers. Given a target value for $P(R_{req} \cap R_{off} \neq \emptyset)$, users compute an appropriate fanout value themselves.

Malicious Matchmakers. Equation 1 assumes that all matchmakers in the set $R_{req} \cap R_{off}$ follow the protocol and actually inform order creators when receiving matching orders. This assumption violates our threat model since malicious matchmakers can respond with sub-optimal matches, or not respond with matches at all. Therefore, we modify Equation 1 to account for the situation where a fraction r of all matchmakers is malicious. Intuitively, this situation would require a higher value of f in order to reach at least one honest matchmaker. Given a fraction of malicious matchmakers r, $P(R_{req} \cap R_{off} \neq \emptyset)$ is now equal to:

$$P(R_{req} \cap R_{off} \neq \emptyset) = 1 - \prod_{i=0}^{f-1} \left(\frac{(m - \lfloor (1-r)f \rfloor - i}{m}\right) \tag{2}$$

We show in the next subsection that this is an appropriate modelling of malicious matchmakers. The rationale behind this model is as follows. The quality of matches from malicious matchmakers is likely to be lower compared to those received from honest matchmakers. Therefore, there is a high probability that the effect of malicious matchmakers is negated upon receiving matches from an honest matchmaker, since a user will process the matches of honest matchmakers first. Reaching honest matchmakers in the presence of malicious matchmakers requires a higher fanout value.

When a matchmaker receives an `order` message describing an order O, it matches O with existing orders in its order book with the same type, according to a matching policy (see Section 5). For each order matched against O, the matchmaker constructs a `match` message and sends it to the user that created O (step ③ in Figure 2). A `match` message contains the full specifications of the matched orders, and the network address of the user behind the matched order. This network address is used by the receiver of the `match` message to initiate the order negotiation process with the user behind the matched order.

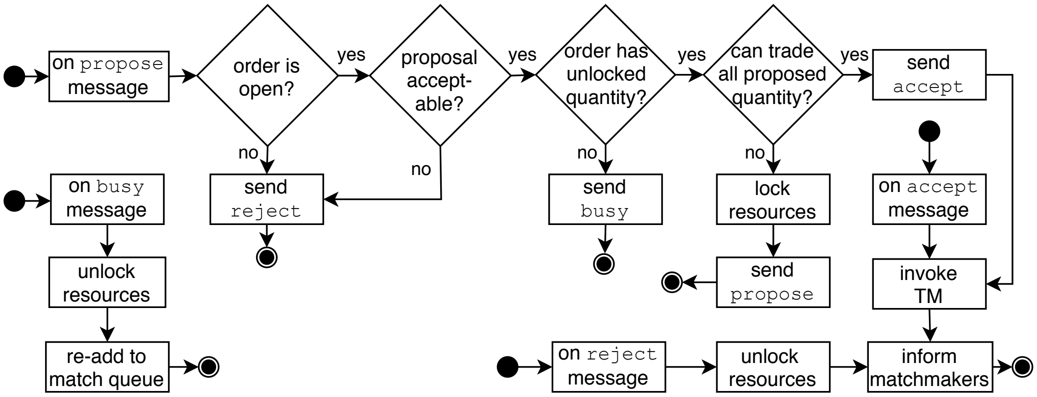


Figure 3. The flow diagram (in UML format) when receiving a message during the negotiation process for a specific order.

4.3 Match Queue

Upon receiving a `match` message from a matchmaker, a user contacts the creator of the matched order and initiates a negotiation process (further discussed in Section 4.4). A potential strategy is that the user immediately contacts another user upon the arrival of a `match` message. This possibly minimizes the time for an order to be completed. However, this strategy leaves the user vulnerable to an attack where a malicious matchmaker is the first to send a specific `match` message to a user, which immediately triggers the negotiation process. The quality of the received match described by the `match` message might be poor and the user might have received a match with a higher quality if it would have waited for additional matches from honest matchmakers.

To address this issue, incoming `match` messages for a specific order are first stored in a *match queue*. When a user receives the first `match` message for one of its offers (or requests), say O, it creates a new match queue for O. Each entry in the match queue of offer (or request) O is a tuple (a, R) where R is a request (or offer) that matches with offer (or request) O. a indicates the number of failed negotiation attempts for R. The value of a is locally tracked by each user. Removing items from the match queue is first based on a (item with a lower value of a have higher priority) and is then based on the quality of the match (the user prioritizes negotiation of matches with higher quality). The quality of a match is an application-specific metric that can be considered as the "distance" between an offer and a request, and is computed by the matching policy. An incoming `match` message can be inserted in multiple match queues, e.g., in the match queues for offers or requests with the same type and similar specifications.

Before a user starts to select items from a match queue, it waits for some duration W_{match}, which we call the *match window*. The value of W_{match} should be carefully considered: a higher value of W_{match} increases the probability of receiving more and better matches but adds to the order completion time since a user has to wait longer before starting order negotiations. Decreasing W_{match}, however, might result in missing better matches. W_{match} also depends on the link latency of the peer-to-peer network. Furthermore, W_{match} is influenced by the trading domain, e.g., passengers in a ride-hailing market can usually tolerate an additional wait time of a few seconds, whereas this increase might be unacceptable when a user quickly wants to buy some assets in response to price fluctuations in an asset market. When the match window expires, a user removes the entry (a, R) with the best quality from the match queue and initiates the order negotiation process with the user that created R.

4.4 Order Negotiation

In MATCH, users negotiate about their orders with other users. This negotiation approach increases resilience against malicious matchmakers since rational users choose to negotiate about the best incoming matches in order to get the best deal. When two negotiating users reach a trade agreement, both users send the agreement to the TM, upon which the trade is executed. This approach addresses question 1 in Section 2.2 and avoids the need for network-wide consensus.

We now elaborate on the negotiation procedure between two users. In the following, we assume that user U_1 created offer O, user U_2 created request R, and these two orders match. Now, U_2 has received a `match` message from a matchmaker, informing U_2 about matching offer O. This puts entry (a, O) in the match queue associated with R. Order negotiation, based on match queue entry (a, O), starts by U_2 locking quantity in request R. How much quantity is locked depends on the available quantities in both O and R. Specifically, U_2 proposes to trade as much available quantity as possible, given the specifications of O and R. Explicitly locking quantity in an order prevents a user from engaging in parallel negotiations for the same resources. MATCH does not enforce the locking of quantity since we assume that rational users will correctly manage their orders. After locking the quantities in R, U_2 sends a `propose` message to U_1, containing the full specifications of R, the identifier of O, and the proposed quantity to trade.

Figure 3 shows the control flow in UML format when a user receives specific messages during the order negotiation process. We elaborate the negotiation process between users U_1 and U_2 according to Figure 3. When U_1 receives a `propose` message from U_2, it first checks if its offer O, which identifier is contained in the `propose` message, is still open. If O is expired, has been cancelled, or has been completed already, U_2 immediately responds with a `reject` message, containing the reject reason. When U_2 receives the `reject` message from U_1, it unlocks the locked quantity for that negotiation and selects the next entry from the match queue of its request R. Since matchmakers might have outdated information about O (e.g., when O has been completed but the matchmaker has not been notified about this event), U_2 forwards the `reject` message to the matchmakers that informed U_2 about the match with O. Matchmakers then update the state of O accordingly when receiving a `reject` message.

If offer O is open, U_1 first determines if the incoming proposal is acceptable. This step depends on the trading domain and specifically on the application-specific data in the order. For example, a matchmaker in a ride-hailing market can establish a valid match between a passenger and driver. However, this match might be unacceptable for one of the matched parties (e.g., when the geographic separation between the parties is too large). If U_1 finds the proposal unacceptable, it sends a `reject` message to U_2.

If the proposal is acceptable, U_1 checks if it has any unlocked quantity in the offer O. If there is no unlocked quantity available for trade, U_1 responds with a `busy` message to U_2, indicating that it currently has no room for negotiation. In this situation, U_1 is already engaged in negotiations for that order with other users. Upon receiving a `busy` message, U_2 unlocks the locked quantity in R and re-adds the entry associated with the failed negotiation to the match queue of R, incrementing a by one. Re-adding this entry to the match queue will cause U_2 to initiate a negotiation with U_1 again later. To prevent a user from immediately retrying a failed negotiation, a user waits a random period (between 1 and 2 seconds) before sending out another `propose` message when processing a match queue entry with $a > 0$.

If offer O has unlocked quantity, U_1 checks whether the full proposed quantity can be traded. If so, U_1 sends an `accept` message to U_2, thereby accepting the proposal of U_2. U_1 also forwards the `accept` message to the matchmakers that proposed the match so they can update the state of this order in their order book. If the proposed quantity is unavailable in O, U_1 makes a counter-proposal by locking as much quantity as possible in O and by sending a `proposal` message back to U_2 with this (lower) quantity. U_1 and U_2 keep sending `propose` messages with differing quantities until one of them responds with either an `accept` or a `reject` message.

It could be that one of the involved parties does not respond during a negotiation, e.g., to deliberately lock quantity in the orders of another user. To address this situation, all outgoing messages during order negotiation have a fixed *negotiation window*, indicated by W_{neg}, after which the user leaves the negotiation. When this window expires, any locked quantity for this negotiation is released and incoming messages regarding the expired negotiation are ignored.

4.5 Match Execution and Order Updates

Upon reaching a trade agreement between two negotiating users, it should be executed by the trading mechanism. To execute a trade agreement, one party sends the `propose` message to the TM and the other party sends the `accept` message to the TM (step ⑧ and ⑨ in Figure 2). Each message contains the digital signature of its creator. The TM executes the trade after having received both these messages. Next, each involved party individually inform the matchmakers that originally received their order about the match execution by sending an `update` message (step ⑩ and ⑪ in Figure 2). This message contains the order with an updated quantity, specifying the new interests of the order creator after the match has been executed. The `update` message contains an order with quantity `0` if it has been completed. Upon receiving an `update` message, matchmakers update the state of the changed order accordingly and remove orders that have been completed.

5 The MATCH Middleware Architecture

We now present the MATCH middleware architecture, visualized in Figure 4. Each user and matchmaker deploy a single instance of the MATCH middleware as a shared runtime library. Communication with the middleware by external applications proceeds through an API, which specifications are included in our open-source implementation. We now elaborate on the components in the MATCH middleware architecture.

Network layer. The network layer passes incoming messages to the MATCH overlay logic and routes outgoing messages to their intended destination. This layer can be implemented using any networking library with support for peer-to-peer communication and authenticated messaging.

Overlay logic. The overlay logic processes incoming messages received by the network layer. It inserts incoming `match` messages in the appropriate match queue, discussed in Section 4.3. It contains policies for order *validation* which specify how the validity of an incoming order with a given type is determined. The validation policy takes into consideration the attributes in the `data` field of the order (see Listing 1), and checks whether they are valid with respect to a trading domain. For example, a validation policy for orders in a ride-hailing market should check whether the latitude and longitude coordinates are included and within a

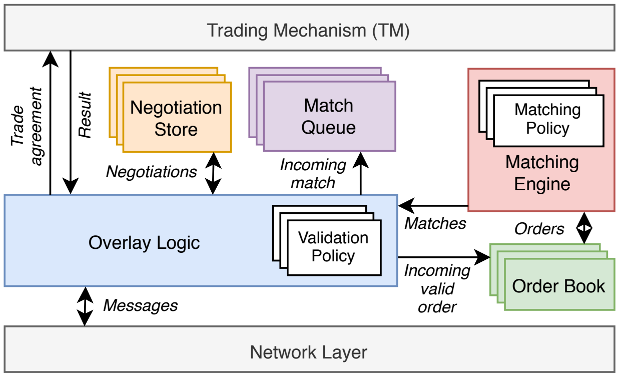


Figure 4. The MATCH middleware architecture.

valid range (-90 to 90). We envision that developers share the implementation of these policies through some distribution medium (e.g., a website). To increase the trustworthiness and security of these policies, the policy implementations should be auditable and attestable by other developers and auditors. These validation policies can then downloaded by users and matchmakers that are interesting in participation within a specific trading domain. We provide developers the means to create custom validation policies, enabling order validation in different trading contexts. Incoming orders deemed invalid by the validation policy are discarded and not processed.

Order books. Each matchmaker can host multiple order books, coloured green in Figure 4, which store orders with differing types. For example, MATCH can maintain an asset trading order book that stores orders to buy or sell Euros for Dollars, and another ride-hailing order book that stores transportation requests and ride offers. Maintaining multiple order books is a key property of MATCH and results in a single and reusable matchmaking solution that can be deployed across different domains.

Matching engine. Valid incoming orders are passed to the matching engine, coloured red in Figure 4. The matching engine attempts to match incoming offers and requests with existing requests and offers, respectively. It contains *matching policies* which predicate whether a specific offer and request match, based on the order type and specifications. For example, matchmaking in a ride-hailing market is often based on the geographic distance between a driver and a passenger. Likewise, asset orders would match if the price of an offer is equal to or lower than the price of a request. Similar to validation policies, matching policies are published on a website or on another public medium, and can be downloaded by interested matchmakers.

Negotiation stores. To correctly process incoming messages from negotiation counterparties, MATCH requires state storage of outgoing messages during order negotiation. This state is stored in a *negotiation store*, coloured yellow in Figure 4. For each negotiation, a new negotiation store is created, a unique identifier is generated, and this identifier is appended to each message associated with this negotiation. Counterparties are required to include the same identifier in response messages. Incoming negotiation messages containing an unknown identifier are discarded by users. Negotiation stores time out after the negotiation window expires, on which the store and its contents are deleted.

Notation	Variable Description
W_{match}	Match window (fixed to 2 seconds)
W_{neg}	Negotiation window (fixed to 5 seconds)
m	Number of matchmakers
f	Order fanout
r	Fraction of malicious matchmakers

Table 1. An overview of the variables used in Section 6.

Trading mechanism. Negotiated trade agreements are passed to the trading mechanism that executes the trade. We consider this component external to MATCH. The trading mechanism notifies the overlay logic when the trade is executed. This notification includes one or more transaction identifiers and a boolean value indicating whether the trade was successful or not. We assume that the trading mechanism provides atomic guarantees: either the full negotiated match is executed or nothing is being executed. This guarantee is, for example, provided by smart contracts, applications that runs on top of a blockchain (also see Section 6.3) [33].

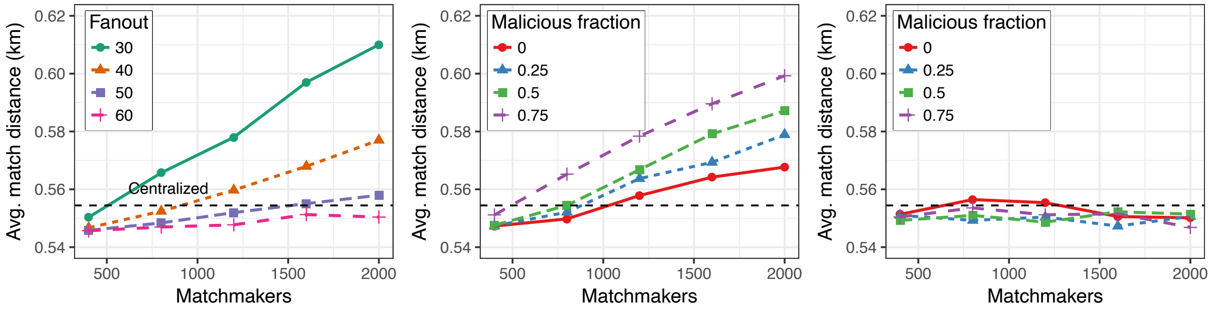
6 Experimental Evaluation

We implement the MATCH protocol and middleware in the Python 3 programming language, spanning a total of 6.511 lines of source code (SLOC), without comments. The implementation uses the `asyncio` library for asynchronous event processing. The network layer is implemented using our networking library, optimized for building peer-to-peer overlay networks and with built-in support for Network Address Translation (NAT) puncturing and authenticated messaging.[1] For efficiency, message exchange between users and matchmakers uses UDP. Order negotiation proceeds using TCP since this flow requires bilateral and reliable message exchange. All software artifacts of MATCH (source code, tests, and documentation) are available online.[2]

In Section 6.1 and 6.2, we subject the MATCH middleware to two workloads for ride-hailing and asset trading, reconstructed from real-world traces. These experiments demonstrates that MATCH maintains high matching quality, is resilient against malicious matchmakers, and is reusable across different trading domains. In Section 6.3, we compare our solution to matchmaking on an Ethereum blockchain and show that MATCH uses considerably less bandwidth and has superior order fulfil latencies.

[1] See https://github.com/tribler/py-ipv8

[2] See https://github.com/Tribler/anydex-core/tree/match-middleware20



(a) Matching quality w. different fanouts f **(b)** Impact of selfish matching ($f = 50$) **(c)** Impact of selfish matching (adaptive f)

Figure 5. The matching quality and impact of selfish matching when executing the ride-hailing workload in MATCH, while varying the number of matchmakers. In Figure (b) and (c), the order fanout f is either fixed ($f = 50$) or adaptive such that $P(R_{req} \cap R_{off} \neq \emptyset) \geq 0.95$.

All experiments are conducted on our nation-wide university cluster, allowing us to run multiple instances of MATCH on different compute nodes [4]. It contains 48 compute nodes, each one equipped with dual 8-core E5-2630v3 CPU and 64GB of memory, running CentOS 6. To account for network latencies, we source a distribution from the PlanetLab latency dataset and uniformly sample from it when sending messages [59]. This also accounts for runtime variability of latency present in real-world networks. Table 1 summarizes the variables that are used in this section. The negotiation window (W_{neg}) is fixed to five seconds, which is well above the highest observed round-trip time in the PlanetLab latency dataset. The match window (W_{match}) is fixed to two seconds. These values should be increased when MATCH is deployed in networks with higher link latency, since it then can take longer for `match` or negotiation messages to arrive.

6.1 Ride-hailing Experiments

Unfair matchmaking in ride-hailing markets is a prominent threat to both drivers and passengers [7, 9]. We leverage the MATCH middleware to devise a decentralized alternative to ride-hailing platforms like Uber and Lyft. In this market, drivers perform matchmaking themselves. The first set of experiments focuses on the matching quality and fairness of MATCH in a ride-hailing environment. The used workload contains temporal information about ride offers and requests created by drivers and passengers, respectively. Each order in the workload has a quantity of one, ensuring that a ride request is matched with at most one ride offer.

Workload specification. The workload is reconstructed from historical traces of taxi rides published by the government of New York [51]. We analyse the traces and subtract 2'100 ride offers and requests during the busiest period in 2015: November 1, 00:59:57 to November 1, 01:01:16 (datasets published after 2015 did not include geographic information on drivers and passengers). We assume a total of 1'100 drivers and 1'000 passengers, to resemble the situation where drivers are waiting idly for passengers. First, drivers indicate their willingness to transport passengers by creating offers containing their waiting location, during 55 seconds (we wait 50 milliseconds between the creation of subsequent ride offers). After this period, passengers submit requests containing their pick-up location, during almost 77 seconds. After all passengers have submitted their request, we leave the experiment running for an additional 60 seconds, to ensure that all requests are matched with an offer. Since the workload does not provide information on the identity of individual passengers or drivers, it is assumed that each passenger creates one request throughout the experiment. This assumption does not lead to skewed results since a passenger does not create multiple ride requests within short time [41].

For the this workload, we implement the matching policy such that it minimizes the distance between passengers and drivers, to reduce waiting times for passengers. Specifically, the policy computes the geographic (haversine) distance between the locations included in offers and requests. In this market, we define the matching quality as the average distance between matched passengers and drivers, which we also call the *match distance*. This quality metric is also used by related work on matchmaking in ride-hailing markets [41].

Matching quality. We first quantify the matching quality of MATCH under the ride-hailing workload when increasing the number of matchmakers for different values of the order fanout, see Figure 5a. The horizontal axis shows the number of matchmakers (m) and the vertical axis denotes the average match distance. For this experiment, all matchmakers act honest and execute the same matching algorithm (in other words, $r = 0$). As a baseline, we use the matching quality of a centralized matchmaking approach where a single server matches incoming orders in a FIFO manner (following the model in Figure 1). This centralized approach results in an average match distance of 0.544km, and is indicated with a dashed horizontal line in the graphs of Figure 5. Figure 5a shows that the average match distance increases when there are more matchmakers under a fixed order fanout. Also, the match distance increases when the order fanout decreases. In particular, It becomes less likely that (good) matches for offers and requests are found when either the number of matchmakers increases or the order fanout decreases. The match distance increases significantly when $f = 30$ and m increases. E.g., for $m = 2'000$, the average match distance increases to 0.607km, 9.5% higher compared to the baseline.

Remarkably, for lower values of f and m, *MATCH outperforms the performance of centralized matchmaking, in terms of matching quality.* We explain this behaviour as follows. With our ride-hailing workload, centralized matchmaking can immediately match a ride request with a ride offer. The overall match quality, however, might be improved when batching incoming ride requests, since it could have been better to assign an already-matched driver to a passenger that created its request later during the experiment. Call markets, for example, operate in batches, where incoming orders are first aggregated over time and then matched at predetermined time intervals. In MATCH, users aggregate potential matches during the match window, W_{match}, resembling this behaviour. Therefore, the matching quality in MATCH can exceed that of centralized FIFO matchmaking because of match aggregation by users, at the cost of a larger order fulfil time.

Impact of selfish matching. We show how selfish behaviour of malicious matchmakers impacts the matching quality. We model a malicious matchmaker as a driver that matches an incoming ride request from a passenger with its own service offer first. This captures the economic incentive of drivers to prioritize their own ride services.

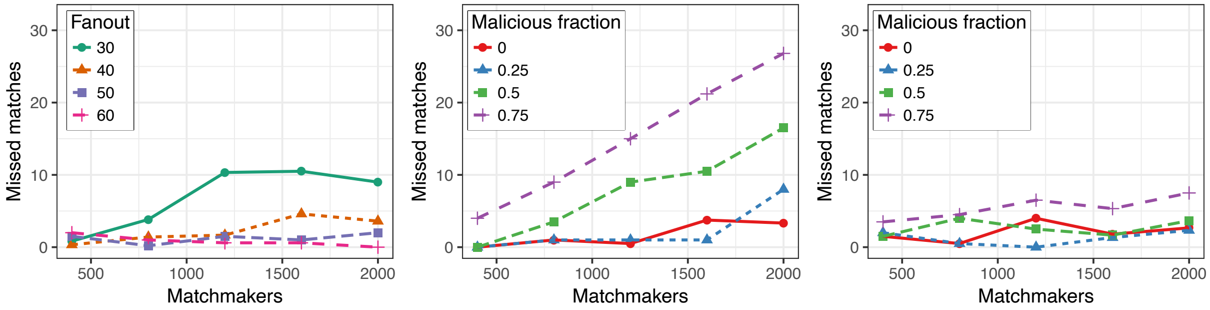
Figure 5b shows the average match distance under a fixed order fanout ($f = 50$) when increasing the number of matchmakers. We vary r, up to 75% of all matchmakers ($r = 0.75$). Figure 5b shows that increasing both m and r has a negative impact on the matching quality in MATCH. The problem is that a malicious matchmaker matches the requests of passengers with its own offer, while it likely would have established a better match if the matchmaker would have been honest. Therefore, we also consider an adaptive order fanout, based on the values of both r and m. Specifically, f is fixed to the lowest integer value such that $P(R_{req} \cap R_{off} \neq \emptyset) \geq 0.95$. Figure 5c visualizes these results with an adaptive order fanout. The order fanout is 63 with $m = 2'000$ and $r = 0$. Formula 2 describes that when 50% of the matchmakers prioritize their own ride offer ($r = 0.5$), the order fanout increases to 78. Figure 5c shows that the average match distance remains largely the same, even when increasing the total number of matchmakers. These results show that in a network with 2'000 matchmakers, by increasing the order fanout by only 15, MATCH can tolerate with 50% of all matchmakers acting maliciously and still produce a matching quality that is on par with the situation where all matchmakers are honest. In practice, the exact value of r is not known a-priori and MATCH developers should therefore fix the order fanout to account for an upper bound for the malicious fraction (e.g., many BFT consensus algorithms tolerate up to $r = \frac{1}{3}$ [12]).

6.2 Asset Trading Experiments

We now evaluate MATCH in an asset trading domain. Driven by the popularity of blockchain-based assets, major peer-to-peer markets have emerged to facilitate cryptocurrency exchange between traders [6]. MATCH enables traders to perform matchmaking of orders themselves in a fair manner, without entrusting their orders to a market operator. Unlike our previous experiments, the workload used in our upcoming experiments involves offers and requests that be partially fulfilled and are commonly cancelled. We conduct the same matching quality and fairness experiments described in Section 6.1 with an asset trading workload.

To the best of our knowledge, there is no standardized definition for the matching quality of orders with partial fulfilment. Therefore, after each experiment with the asset trading workload, we determine the matching quality as follows: all orders that are not cancelled or fulfilled are added to a single order book, starting with the first order created. When adding these orders to the order book, we sum the number of matches returned by the matching engine, which yields our matching quality. Intuitively, this definition indicates how many additional matches a central matchmaker would have found if it had knowledge of all open orders. By definition, the matching quality of centralized matchmaking is zero and therefore optimal with FIFO order matching. In the worst case, our middleware would have missed 6'450 matches, which is the situation where no matchmaker performs any matching. When running the asset trading workload, the matching engine matches orders according to the *price-time* matching policy [35].

Workload specification. The asset trading workload contains buy and sell orders that have been published on the blockchain ledger of BitShares [46]. BitShares is a blockchain-based decentralized exchange where users can create, issue and trade digital assets. New orders are submitted to dedicated validator nodes, which include incoming orders in a block on the blockchain. We have analysed the entire BitShares blockchain since its inception and extracted all buy and sell orders, and order cancellations. To generate load on our system, we determined when most orders were created for five minutes. The result is a workload with 942 order cancellations, 12'253 offers and 3'342 requests involving 121 unique asset types. On average, traders create 52 new orders every second. Since our dataset does not contain temporal information on order creation and cancellation, we assume that each order is uniformly created in the time interval between the last block and the block that contains this specific



(a) Matching quality w. different fanouts f **(b)** Impact of selfish matching ($f = 50$) **(c)** Impact of selfish matching (adaptive f)

Figure 6. The matching quality and impact of selfish matching when executing the ride-hailing workload in MATCH, while varying the number of matchmakers. In Figure (b) and (c), the order fanout f is either fixed ($f = 50$) or adaptive such that $P(R_{req} \cap R_{off} \neq \emptyset) \geq 0.95$.

order. We believe this approximates the actual creation timestamp of the order and that this does not skew the experiment results. Again, there is a 60 seconds experiment cooldown period after the creation of the last order.

Matching quality. Figure 6a shows the matching quality while varying the number of matchmakers and order fanout. By definition, a centralized approach to matchmaking would not miss any match. Similar to the matching quality experiment with the ride-hailing workload (see Figure 5a), matching quality decreases when there are more matchmakers and the order fanout is static. It particularly interesting to observe how even a relative low order fanout of 30 results in less than ten missed matches on average (only 0.61% of the maximum number of missed matches). Further analysis of the workload reveals that various users create multiple orders for the same asset pair within short time. Therefore, `match` messages for such orders received are inserted in multiple match queues, and thus re-used. Users creating multiple smaller orders with similar specifications are reaching more matchmakers and can potentially negotiate better matches.

Impact of selfish matching. We demonstrate the effect of malicious matchmakers on the matching quality in our asset trading workload, both with a fixed and adaptive order fanout. Under the asset trading workload, we model a malicious matchmaker as a node that purposefully does not inform the party behind an incoming order about the match with the best price. Essentially, a malicious matchmaker "hides" order book entries from the order creator.

Figure 6b shows the number of missed matches with $f = 50$ when increasing m and varying the r. More matchmakers negatively impacts the matching quality, although its effect is relatively minor. In particular, even with $r = 0.5$ and $m = 2'000$, MATCH only misses less than ten matches on average. We repeat the experiment while adapting the order fanout such that $P(R_{req} \cap R_{off} \neq \emptyset) \geq 0.95$, see Figure 6c. It shows that for all settings, MATCH only misses under ten matches on average.

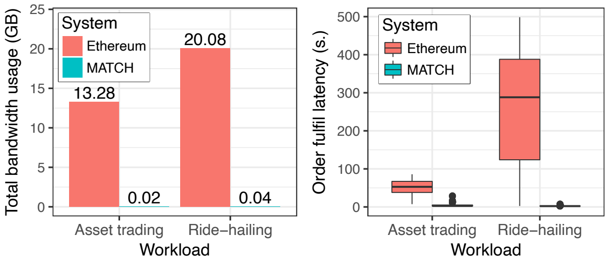
6.3 Comparison with On-chain Matchmaking

We compare the bandwidth usage and order fulfil latencies of MATCH with that of matchmaking on an Ethereum blockchain, using both the ride-hailing and asset trading workloads. Ethereum is the most mature blockchain platform that enables the execution of generic-purpose smart contracts, and is the most used platform to deploy smart contracts in general [58].

We setup a private Ethereum network consisting of 400 instances running `geth`, an Ethereum client written in Go.[3] Ethereum uses a Proof-of-Work consensus mechanism in which participants, also called miners, compete to include transactions on the blockchain. Specifically, each miner continuously solves an algorithmic puzzle and the first miner to find a valid solution to the puzzle, can extend the blockchain with one block with transactions. We fix `geth` such that each instance mines on at most one CPU core. We fix the gas limit (indicating the maximum amount of computation that can be done within a block) to 10'000'000, in line with the public Ethereum network. To accurately compare the performance of MATCH and Ethereum, we run both workloads in MATCH with 400 instances, and adjust our workload accordingly. We fix $m = 400$, $f = 30$ and $r = 0$. Since a smart contract enforces the correct execution of a particular matching policy, we run MATCH with 400 honest matchmakers.

Smart contracts. For both workloads, we implement a smart contract in the Solidity programming language. The smart contract for the ride-hailing workload maintains two lists containing open offers and requests. Submission of a new ride offer and request is done by issuing a transaction with geographic information that invokes the `ride_offer`

[3] See https://github.com/ethereum/go-ethereum



(a) Total bandwidth usage **(b)** Order fulfil latency

Figure 7. The total bandwidth usage and the distribution of order fulfil latencies of on-chain matchmaking on Ethereum and in MATCH, under the ride-hailing and asset trading workloads.

and `ride_request` methods in the smart contract, respectively. Invocation of these methods triggers a loop through the list of active offers or request, and finds the matching order that minimizes the distance between the passenger and driver. The algorithmic complexity when matching a single offer and request is $O(n)$ where n is the number of open requests and offers, respectively. To avoid computationally expensive trigonometry operations when computing the Haversine distance, latitude and longitude coordinates are projected to Universal Transverse Mercator (UTM) coordinates and the Manhattan distance is used as a norm in the smart contract. This results in an accuracy loss of only 0.35%.

For the asset trading workload, we adopt an existing and deployed order book implementation.[4] This smart contract implements a market to trade digital tokens that reside on the Ethereum blockchain. Orders are bundled in a limit order book and organized in distinct price levels. This allows for a strategic search for order matches and avoids the need for a full linear scan through all offers and requests. This order book organization is predominantly used by financial exchanges. To quantify the overhead of order matchmaking, we remove the operation that transfers token ownership after matching from the smart contract but leave the implemented price-time matchmaking logic intact.

Bandwidth usage. We measure the aggregated bandwidth usage of all instance of MATCH and Ethereum, see Figure 7a. Ethereum requires over 20GB of network traffic for the ride-hailing workload. In comparison, MATCH uses dramatically less bandwidth compared to Ethereum-based matchmaking. MATCH only requires 41.6MB of aggregate network traffic under the ride-haling workload, and 20.7MB under the asset trading workload. The high bandwidth usage of Ethereum is a direct consequence of the full replication of state. Specifically, each transaction and block is disseminated to all active Ethereum instances, resulting in a significant amount of network traffic.

Order fulfil latencies. In Figure 7b, we show the time it takes to complete orders in MATCH and Ethereum, for both workloads. Specifically, this is the time between order creation and order fulfilment. For the ride-hailing workload, we only consider the completion time of requests made by passengers, since drivers are waiting for incoming requests. Figure 7b shows that the average order completion time of MATCH under the ride-hailing workload is 2.46 seconds and increases under the asset trading workload to 5.02 seconds. Since users aggregate `match` messages during the match window, the order fulfil latency in MATCH is at least W_{match} (which is fixed to two seconds in our experiments). In comparison, the average order fulfil latency in Ethereum is 258.2 and 53.6 seconds under the ride-hailing and asset trading workload, respectively. We argue that in a ride-hailing market, the latencies experienced when performing matchmaking on an Ethereum blockchain would be unacceptable for both passengers and drivers.

Ethereum transaction pool. To further analyse the large differences in order fulfil latencies between MATCH and Ethereum, we visualize the size of the Ethereum transaction pool (as maintained by a single Ethereum instance) during the experiment in Figure 8. This figure shows the time into the experiment on the horizontal axis and the number of transactions in the pool on the vertical axis. The transaction pool contains transactions that are not yet included in a block on the blockchain by a miner. Note how Ethereum becomes congested under the asset trading workload quickly after starting the experiment. Around 70 seconds into the asset trading experiment, the transaction pool contains 1'713 unconfirmed transactions that are buying or selling assets. 285 seconds after the start of this experiments, all orders are included in a block on the Ethereum blockchain.

The ride hailing experiment starts by drivers submitting ride offers to Ethereum instances. All ride offers are included on the Ethereum blockchain after 58 seconds into the experiment. Passengers start to submit ride requests 100 seconds into the experiment. Similar to the asset trading workload, the size increase of the Ethereum transaction pool shows that

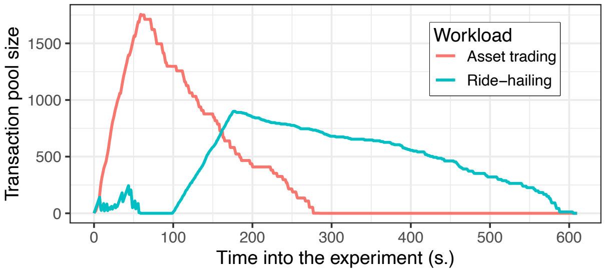


Figure 8. The size of the transaction pool during our Ethereum experiments, under the ride-hailing and asset trading workloads.

[4] See https://github.com/makerdao/maker-otc/tree/master/src

the blockchain is unable to handle the load of incoming transaction, causing congestion. 180 seconds into the experiment, the number of unconfirmed transactions decreases. Further inspection of the blockchain reveals that only around ten transactions with a ride request are included in each block. We explain this behaviour as follows. In Ethereum, the sum of gas usage of all transactions in a block cannot exceed 10’000’000 gas. The gas cost of matching ride requests scales with the number of open ride offers and decreases during the experiment since there are less offers to compare with. Note how after 500 seconds into the ride-hailing experiment, the number of unconfirmed transactions decreases quickly.

7 Related Work

Matchmaking (or brokering) is a core concept in publish/-subscribe (Pub/Sub) architectures. In centralized Pub/Sub architectures, a single server brokers incoming messages between publishers and subscribers [10]. Decentralized approaches either flood events through the entire network, or route these events based on their topic or content [5, 11]. In contrast to Pub/Sub systems, MATCH does not ensure that events (orders) are eventually delivered to all subscribers (matchmakers).

Resource allocation, the assignment of compute resources to incoming jobs, also requires matchmaking [15]. Most work on resource allocation aims to find an optimal assignment between resources and jobs, whereas our work focuses on establishing any match [32]. In this context, we identify two matchmaking approaches described in literature. The first approach is to use market mechanisms that coordinate the matchmaking process, e.g., by a continuous double auction mechanism [8, 26, 38]. Market-based matching approaches increase allocation efficiency but compromise on computational efficiency since it requires synchronization mechanisms. The second approach to matchmaking is to deploy one or more dedicated (centralized) brokers [19, 22, 43].

Motivated by the scalability and load balancing issues of centralized matchmaking, various researchers explored the usage of multiple, independent matchmakers [1, 2, 20, 48]. Matchmakers usually operate within their own administrative domain, acting as a broker for a specific set of nodes. The work of Shafran et al. evaluates a distributed matchmaking model where orders are cached by intermediate agents [47]. Their work, however, only considers one-to-one matching.

With the proliferation of blockchain-based tokens, there have been various proposals for matchmaking architectures that complement decentralized exchanges. These architectures aim to avoid transaction fees and expensive on-chain matchmaking by relying on an off-chain order matching service, and on-chain order execution. IDEX, an Ethereum-based decentralized exchange, uses a centralized server for order matchmaking [31]. In AirSwap, *indexers* mediate trade between makers, nodes who create an order, and takers, who fulfil existing orders [37]. In contrast to MATCH, a user can only send a new order to a single AirSwap indexer. The 0x protocol uses a similar matchmaking approach since traders send a new order to exactly one matchmaker [57]. The Loopring protocol is similar to the decentralized matching model of MATCH since traders submit orders to one or more *relay nodes* [56]. The protocol description, however, lacks details on the dissemination strategy of orders to matchmakers.

Auctions are related to order matchmaking since they also provide a mechanism to allocate resources from sellers to buyers. PeerMart is a decentralized auction mechanism that uses sets of distributed brokers [28]. There have been various proposals to run Ethereum-based auctions while preserving the privacy of bidders [23, 24]. Yet, auctions and order matchmaking are different economic primitives with differing goals. In contrast to order matchmaking, timeframes (and time limitations) are critical in auctions. Furthermore, auctions have higher security requirements and need (time-bounded) coordination amongst participants, e.g., to determine the winning bidder.

8 Conclusions

We have presented MATCH, a decentralized middleware for fair matchmaking in peer-to-peer markets. Our work addresses fairness concerns associated with the use of in-house, proprietary solutions controlled by a market operator. In the MATCH protocol, users send new orders to a small, random subset of matchmakers, which inform users about potential matches. Users then engage in peer-to-peer negotiation about matches with other users. This approach makes MATCH resilient against matchmakers who deviate from a standard matching policy. We have devised the MATCH middleware architecture, suitable for deployment in a WAN environment. We have experimentally proven resistance against malicious matchmakers in a ride-hailing and asset trading domain, showing that MATCH still establishes high-quality matches. Our comparison experiments have showed that the resource usage of MATCH is considerably lower compared to that of matchmaking on an Ethereum blockchain.

References

[1] Tariq Abdullah et al. 2010. Effect of the degree of neighborhood on resource discovery in ad hoc grids. In *International Conference on Architecture of Computing Systems*. Springer, 174–186.

[2] Abdulrahman A Azab et al. 2008. An adaptive decentralized scheduling mechanism for peer-to-peer desktop grids. In *2008 International Conference on Computer Engineering & Systems*. IEEE, 364–371.

[3] Eduardo M. Azevedo and E. Glen Weyl. 2016. Matching markets in the digital age. *Science* 352, 6289 (2016), 1056–1057.

[4] Henri Bal, Dick Epema, Cees de Laat, Rob van Nieuwpoort, John Romein, Frank Seinstra, Cees Snoek, and Harry Wijshoff. 2016. A medium-scale distributed system for computer science research: Infrastructure for the long term. *Computer* 49, 5 (2016), 54–63.

[5] Guruduth Banavar et al. 1999. An efficient multicast protocol for content-based publish-subscribe systems. In *Proceedings. 19th IEEE*

International Conference on Distributed Computing Systems (Cat. No. 99CB37003). IEEE, 262–272.

[6] Iddo Bentov et al. 2019. Tesseract: Real-time cryptocurrency exchange using trusted hardware. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 1521–1538.

[7] Eszter Bokányi and Anikó Hannák. 2020. Understanding Inequalities in Ride-Hailing Services Through Simulations. *Scientific Reports* 10, 1 (Dec. 2020), 6500. https://doi.org/10.1038/s41598-020-63171-9

[8] Rajkumar Buyya et al. 2002. Economic models for resource management and scheduling in Grid computing. *Concurrency and Computation: Practice and Experience* 14 (2002), 1507–1542.

[9] Ryan Calo and Alex Rosenblat. 2017. The taking economy: Uber, information, and power. *Colum. L. Rev.* 117 (2017), 1623.

[10] Antonio Carzaniga et al. 2001. Design and evaluation of a wide-area event notification service. *ACM Trans. Comput. Syst.* 19 (2001), 332–383.

[11] Antonio Carzaniga et al. 2004. A routing scheme for content-based networking. In *IEEE INFOCOM 2004*, Vol. 2. IEEE, 918–928.

[12] Miguel Castro, Barbara Liskov, et al. 1999. Practical Byzantine fault tolerance. In *OSDI*, Vol. 99. 173–186.

[13] Yatin Chawathe et al. 2003. Making gnutella-like p2p systems scalable. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*. 407–418.

[14] Pepper D. Culpepper and Kathleen Thelen. 2020. Are We All Amazon Primed? Consumers and the Politics of Platform Power. *Comparative Political Studies* 53, 2 (2020), 288–318.

[15] Karl Czajkowski et al. 1998. A resource management architecture for metacomputing systems. In *JSSPP*. Springer, 62–82.

[16] Philip Daian et al. 2019. Flash Boys 2.0: Frontrunning, Transaction Reordering, and Consensus Instability in Decentralized Exchanges. *arXiv e-prints* (2019). arXiv:1904.05234

[17] Ernesto Damiani, S De Capitani Di Vimercati, and Pierangela Samarati. 2003. Managing multiple and dependable identities. *IEEE Internet Computing* 7, 6 (2003), 29–37.

[18] John R Douceur. 2002. The sybil attack. In *International workshop on peer-to-peer systems*. Springer, 251–260.

[19] B Pour Ebrahimi, K Bertels, S Vassiliadis, and K Sigdel. 2004. Matchmaking within multi-agent systems. *Proceeding of ProRisc-2004* (2004).

[20] Janick Edinger et al. 2016. Decentralized scheduling for tasklets. In *Proceedings of the Posters and Demos Session of the 17th International Middleware Conference*. 7–8.

[21] Shayan Eskandari et al. 2019. SoK: Transparent Dishonesty: front-running attacks on Blockchain. *CoRR* abs/1902.05164 (2019).

[22] James Frey et al. 2002. Condor-G: A computation management agent for multi-institutional grids. *Cluster Computing* 5, 3 (2002), 237–246.

[23] Hisham S Galal and Amr M Youssef. 2018. Succinctly verifiable sealed-bid auction smart contract. In *Data Privacy Management, Cryptocurrencies and Blockchain Technology*. Springer, 3–19.

[24] Hisham S Galal and Amr M Youssef. 2018. Verifiable sealed-bid auction on the ethereum blockchain. In *International Conference on Financial Cryptography and Data Security*. Springer, 265–278.

[25] Arthur Gervais, Ghassan O Karame, Karl Wüst, Vasileios Glykantzis, Hubert Ritzdorf, and Srdjan Capkun. 2016. On the security and performance of proof of work blockchains. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 3–16.

[26] Jacek Gomoluch et al. 2003. Market-based Resource Allocation for Grid Computing: A Model and Simulation.. In *Middleware*. 211–218.

[27] Aniko Hannak et al. 2014. Measuring Price Discrimination and Steering on E-Commerce Web Sites. In *ICM*. 305–318.

[28] D Hausheer and Burkhard Stiller. 2005. Decentralized auction-based pricing with peermart. In *2005 9th IFIP/IEEE International Symposium on Integrated Network Management, 2005. IM 2005*. IEEE, 381–394.

[29] Aljosha Judmayer et al. 2019. Pay-To-Win: Incentive Attacks on Proof-of-Work Cryptocurrencies. *IACR* 2019 (2019), 775.

[30] Aashish Kolluri et al. 2018. Exploiting The Laws of Order in Smart Contracts. *CoRR* abs/1810.11605 (2018).

[31] Aurora Labs. 2018. Aurora: A Decentralized Financial Institution Utilizing Distributed Computing and the Ethereum Network.

[32] Simone A Ludwig et al. 2012. Matchmaking in multi-attribute auctions using a genetic algorithm and a particle swarm approach. In *New Trends in Agent-Based Complex Automated Negotiations*. 81–98.

[33] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 254–269.

[34] Arvind Malhotra and Marshall Van Alstyne. 2014. The dark side of the sharing economy… and how to lighten it. *Commun. ACM* 57, 11 (2014), 24–27.

[35] Vasilios Mavroudis and Hayden Melton. 2019. Libra: Fair order-matching for electronic financial exchanges. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*. 156–168.

[36] Petar Maymounkov and David Mazieres. 2002. Kademlia: A peer-to-peer information system based on the xor metric. In *International Workshop on Peer-to-Peer Systems*. Springer, 53–65.

[37] Oved Michael and Mosites Don. 2017. Swap: A Peer-to-Peer Protocol for Trading Ethereum Tokens. https://swap.tech/whitepaper/

[38] Marian Mihailescu and Yong Meng Teo. 2010. A distributed market framework for large-scale resource sharing. In *European Conference on Parallel Processing*. Springer, 418–430.

[39] Alexander Mühle, Andreas Grüner, Tatiana Gayvoronskaya, and Christoph Meinel. 2018. A survey on essential components of a self-sovereign identity. *Computer Science Review* 30 (2018), 80–86.

[40] OECD. 2019. *An Introduction to Online Platforms and Their Role in the Digital Transformation*. 216 pages. https://doi.org/10.1787/53e5f593-en

[41] Anh Pham et al. 2017. Oride: A privacy-preserving yet accountable ride-hailing service. In *USENIX Security*. 1235–1252.

[42] PWC. 2015. *Consumer Intelligence Series: The Sharing Economy*. Technical Report.

[43] Rajesh Raman et al. 1998. Matchmaking: Distributed resource management for high throughput computing. In *Proceedings of The 7th International Symposium on High Performance Distributed Computing*. IEEE, 140–146.

[44] Juliet Schor et al. 2016. Debating the sharing economy. *Journal of Self-Governance and Management Economics* 4, 3 (2016), 7–22.

[45] Zvi Schreiber. 2020. k-Root-n: An Efficient Algorithm for Avoiding Short Term Double-Spending Alongside Distributed Ledger Technologies such as Blockchain. *Information* 11, 2 (2020), 90.

[46] Fabian Schuh and Daniel Larimer. 2017. Bitshares 2.0: general overview. http://docs.bitshares.org/downloads/bitshares-general.pdf

[47] Victor Shafran et al. 2008. Towards bidirectional distributed matchmaking. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems-Volume 3*. International Foundation for Autonomous Agents and Multiagent Systems, 1437–1440.

[48] K Sigdel et al. 2005. A framework for adaptive matchmaking in distributed computing. In *In proceeding of GRID Workshop*, Vol. 224.

[49] Quinten Stokkink, Dick Epema, and Johan Pouwelse. 2020. A Truly Self-Sovereign Identity System. *arXiv preprint arXiv:2007.00415* (2020).

[50] Hemang Subramanian. 2017. Decentralized blockchain-based electronic marketplaces. *Commun. ACM* 61, 1 (2017), 78–84.

[51] N TLC. 2017. Nyc taxi and limousine commission (tlc) trip record data.

[52] David Trastour et al. 2002. Semantic web support for the business-to-business e-commerce lifecycle. In *Proceedings of the 11th international conference on World Wide Web*. 89–98.

[53] Daniel J Veit. 2003. *Matchmaking in electronic markets: An agent-based approach towards matchmaking in electronic negotiations*. Vol. 2882. Springer.

[54] Daniel J Veit et al. 2002. Multi-dimensional matchmaking for electronic markets. *Applied Artificial Intelligence* 16, 9-10 (2002), 853–869.

[55] Marko Vukolić. 2015. The quest for scalable blockchain fabric: Proof-of-work vs. BFT replication. In *International Workshop on Open Problems in Network Security*. Springer, 112–125.

[56] Daniel Wang et al. 2018. Loopring: A decentralized token exchange protocol. (2018). https://github.com/Loopring/whitepaper/blob/master/en_whitepaper.pdf

[57] Will Warren and Amir Bandeali. 2017. 0x: An open protocol for decentralized exchange on the Ethereum blockchain. https://github.com/0xProject/whitepaper

[58] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32.

[59] Rui Zhu et al. 2016. Network latency estimation for personal devices: A matrix completion approach. *IEEE/ACM Transactions on Networking* 25, 2 (2016), 724–737.