PeerMatcher: Decentralized Partnership Formation

Nicolae Vladimir Bozdog, Spyros Voulgaris, Henri Bal, Aart van Halteren
Department of Computer Science
VU University Amsterdam
Amsterdam, The Netherlands
{n.v.bozdog, spyros.voulgaris, h.e.bal, a.t.van.halteren}@vu.nl

***Abstract*—This paper presents PeerMatcher, a fully decentralized algorithm solving the k-clique matching problem. The aim of k-clique matching is to cluster a set of nodes having pairwise weights into k-size groups of maximal total weight. Since solving the problem requires exponential time, PeerMatcher employs a novel set of heuristics that aim at converging to the optimal grouping while keeping the associated time and computational complexity low. A key feature is the use of peer-to-peer communication. An extensive evaluation of PeerMatcher demonstrates its accuracy, efficiency, and scalability.**

I. Introduction

In a world that starts to look more and more like the Internet of Things envisaged by Kevin Ashton in 1999 [1], where tangible items, electronic items, or even human beings are interconnected in a huge network, the need to group these entities by various criteria arises. Most often, it is desired to form groups based on *similarity*. This seems to be the natural way to organize and structure entities. Marketing aims at grouping people based on shopping behavior, libraries group books of similar thematic entities together, insurance companies try to identify groups of people with the same characteristics, while, humans—above all—tend to socialize with people of similar interests, same age, akin social status, or common origins. This kind of similarity-based clustering or grouping is so ubiquitous that we often do not even notice it.

It is not uncommon, however, for *dissimilarity* to be the key for other types of grouping. Consider, for instance, a team for a survival game aiming at combining people with highly diverse, *complementary* skills, to maximize the team's potential. Or, in the corporate world, people (or companies) with complementary expertise (e.g., engineers, lawyers, theoreticians, marketeers, financial analysts) need to collaborate to achieve a common goal.

Finally, there are other cases where the grouping and self-organization of people in teams is driven by arbitrary, diverse criteria. For instance, some team may be attractive to different members for different reasons, such as, for a competent working environment, for an appealing salary, or for a great office location, to name a few.

What is common in all aforementioned examples, is that we have entities that want to form partnerships with each other, each one aiming at improving its benefit from teaming up. Ironically, the spread of the Internet has not made the situation any better. Often, the number of options is so large, that optimizing the selection of small partnerships becomes a nontrivial task.

If we limit the size of partnerships to a fixed value k, all these problems become instances of the *weighted k-clique matching problem* [2]. In this problem, the set of entities that have to be matched is represented as a graph in which every pair of nodes has a *weight*, reflecting the two nodes' mutual interest to become partners. The goal is to group all nodes in a number of non-intersecting k-cliques (i.e., complete subgraphs of k vertices), in such a way that each node finds itself in a clique of the maximum possible aggregate weight.

This can be seen as a generalization of the *maximum weighted matching problem* in a graph, where the goal is to form k-cliques for k=2. While finding a maximum weighted 2-clique matching can be done in polynomial time [3], finding a maximum weighted k-clique matching for $k \geq 3$ becomes an NP-hard problem [4], which can't be solved in a reasonable amount of time by a serial algorithm. To reduce the computational burden, we can resort to two common practices: using heuristics and distributing the computational load among many computing nodes. While the usage of heuristics is a typical way of reducing the search space and approximating the solution, distributing the computation is often a complex task with many levels of granularity. Designing a distributed algorithm ranges from having separate processes running in parallel on different computing cores of the same machine to having different machines spread all over the world that work collaboratively in order to attain a common goal.

In this work, we focus on the case where it is desirable to have the computation spread across processes at a coarse-grained level. In our model, each process represents a node in the graph that aims to partner with $k-1$ other nodes such that the quality of its partnership is maximal, without being concerned with the quality of other partnerships. It has been shown in [2] that having the nodes adopting such a selfish strategy leads to a weighted *k-clique matching* whose weight is off by at most a factor k. Moreover, the nodes should be able to organize themselves into groups without the need of a central authority. This way the fairness and privacy concerns of having all the information about the nodes stored and managed by a centralized server are eliminated.

In this paper we present a decentralized protocol, called PeerMatcher, that approximates the solution of the *weighted k-clique matching problem* by using a new heuristic, called *clique swapping*. We show that our protocol finds a near-

optimal matching ten times faster than existing approaches, while keeping the computational and communication costs low. Our experiments also show that our protocol scales well with an increase in clique and/or network size.

The rest of the paper is structured as follows. Section II gives an overview of the existing research that is related to the weighted k-clique matching problem, while Section III presents our system model. Section IV explains the PeerMatcher protocol, starting from a high-level overview followed by a close look at its details. Section V presents an extensive evaluation and comparison with the state of the art protocol for k-clique matching, and Section VI concludes.

II. Related Work

Previous research was focused more on finding matchings in graphs, which are subsets of edges without common vertices. In the case of unweighted graphs, it is particularly interesting to find maximum matchings, which are matchings with the largest number of edges. Such problems can be solved in polynomial time [5], [6].

In weighted graphs, it is of interest to find the *maximum weighted matching*, which is defined as a matching where the sum of the values of the edges in the matching have a maximal value. Finding such a matching is known as the *assignment problem* and it can be solved in polynomial time using the algorithm by Gabow [3], that runs in $O(|V|(|E|+|V|log|V|))$. The assignment problem is a particular case of the more general problem of finding *H-matchings* [7] in a graph. An H-matching is a subset of nonadjacent subgraphs, where each of them is isomorphic to some given graph H. If H is a complete graph of size k, we obtain the k-clique matching problem that we address in this work.

The first distributed algorithm that computes a weighted matching was proposed by Manne and Mjelde [8]. It finds a $1/2$-approximation to the optimal solution and it takes $O(n)$ rounds to complete. Moreover, their algorithm is self-stabilizing, meaning that the protocol converges to the correct solution regardless of the starting configuration.

Chmielowiec and van Steen [2] extended the previous algorithm to solve the more general problem of weighted k-clique matching. Their approach is also self-stabilizing and computes a solution that is at most a factor k off from the maximum. The same authors propose in [9] a couple of improvements to their algorithm, which are meant to reduce the computational load and decrease the network load. Two major drawbacks of their protocol are the long convergence times and the high number of computations performed by nodes, which makes it impractical to use their protocol for networks of devices with limited computational power (e.g. smartphones). In our work, we address the latter by proposing a novel heuristic, called *cique swapping*, that reduces the complexity of operations performed by each node in a round to $O(n)$. A similar technique was used successfully in the past for balanced graph partitioning [10]. We also introduce a new membership protocol that prevents cliques from overlapping, which greatly improves the convergence speed, as we will see.

III. System Model

We consider a set of N nodes, connected over a routed network infrastructure. Each node is equipped with a unique identifier. The protocol's goal is to group nodes in fixed-sized, non-overlapping partnerships of maximal possible benefit.

Each pair of nodes is assigned a numeric *weight*, that reflects the two nodes' mutual benefit in becoming partners. Weights are fixed, nonnegative, and symmetric. In other words, the benefit a node x perceives in teaming up with node y is the same as the benefit of y teaming up with x. The protocol can be extended to support dynamic and asymmetric weights, but this is out of the scope of this paper. The weights between all pairs of nodes are *globally known* by all nodes a priori.

Nodes try to team up in *k-cliques*, that is, cliques of fixed size k, in a way that maximizes the benefit of their partnerships. A partnership's benefit is expressed as the respective *clique's weight*, which is the average weight among all pairs in a clique. That is, in a k-clique with pairwise weights $w_{i,j}$ with $1{\leq}i{<}j{\leq}k$, the total number of edges is $\frac{k(k-1)}{2}$, and the average clique weight is estimated as:

$$\textit{Clique weight} = \left(\frac{k(k-1)}{2}\right)^{-1} \cdot \sum_{i=1}^{k} \sum_{j=i+1}^{k} w_{i,j}$$

More specifically, each node can only participate in one clique, and is selfishly interested only in maximizing the weight of its own clique.

We consider that nodes are connected over a network that supports routing. That is, *any* node can send a message to *any* other, provided that the sender knows the receiver's address (i.e., IP address and port). Links can experience transient failures, that is, messages may get lost. We also assume that nodes do not crash.

Nodes make use of a *peer sampling service* that provides them with neighbors, picked uniformly at random among all participating nodes. Gossip-based protocols like Cyclon [11] or Newscast [12] can be used to this end. Peer sampling protocols form a fundamental ingredient of many peer-to-peer applications nowadays, they are completely decentralized, and they have shown to be very inexpensive. Most importantly, peer sampling protocols exhibit remarkably self-healing properties, demonstrating high resilience to node and link failures, as well as node churn.

Finally, all communication is asynchronous, and does not require node clocks to be synchronized.

IV. The PeerMatcher Protocol

A. Overview

The suggested algorithm consists of two phases. In the first phase, nodes organize themselves into cliques with k members. They do this by using a simple heuristic based on edges' weights. In the second phase of the protocol, nodes from different complete cliques (cliques with k members) swap places such that the weights of the cliques increase. The two phases are not synchronous, meaning that some nodes can be in phase one, while others are in phase two.

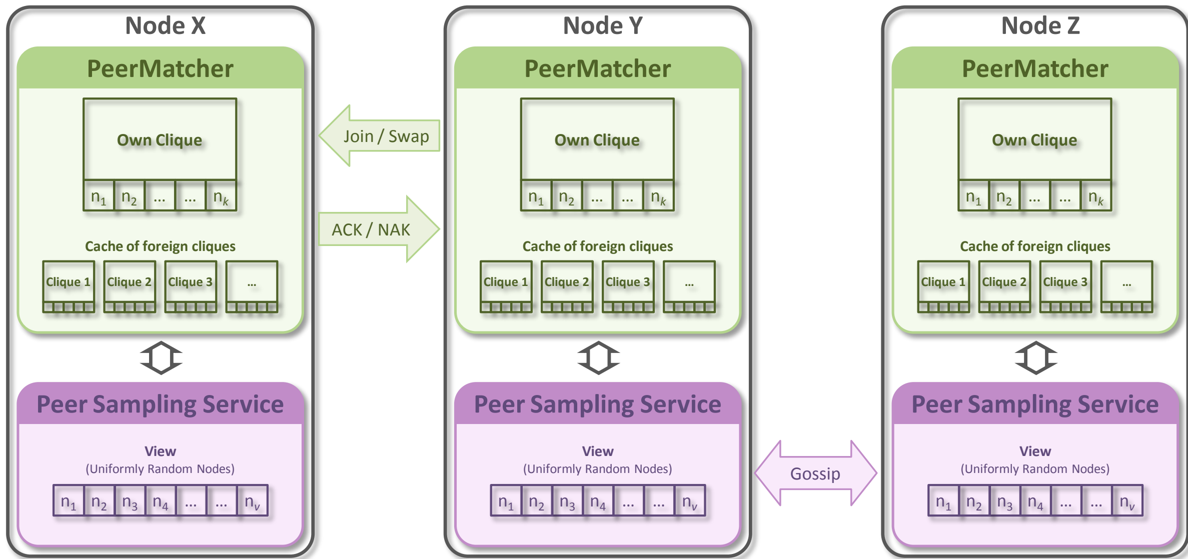


Fig. 1. PeerMatcher architecture

Eventually, the system reaches a stable state, in which all nodes are grouped into cliques of size k and no more swap operations that would improve any of the cliques are possible[1]. Intuitively, the algorithm works similar to bubble-sort. As in bubble-sort numbers in a sequence change places until the sequence becomes sorted, here nodes from different cliques keep swapping places until the weights of the cliques become maximal. We should mention that by using this technique, there is a risk to get stuck in a local optimum, which would prevent the system from converging to a global optimal solution. One way to prevent this is to use a technique called *simulated annealing* [13]. However, since all our experiments gave good results, we present here only the core algorithm, leaving any additional optimizations for future work. Also, we envisage that our protocol will be used in the future in mobile networks to cluster low-powered devices, so we prefer to keep the computation as low as possible.

Maintaining a consistent state for each clique is key to the correct operation of our decentralized protocol. In that direction, PeerMatcher employs a simple synchronization mechanism. Each clique appoints a *leader*, which is unambiguously determined as the node with the highest ID among the clique members. The role of the leader is to store the state of the clique (the list of members) and to synchronize all the operations that could change it. All requests that could change a clique's state have to be approved by its leader. When the leader of a clique changes (this can happen for example when a new node is accepted to a clique and its ID is the highest), the state of the clique is safely transferred from the old leader to the new one. We explain later how this is done.

Without this point of synchronization, different types of inconsistencies may occur. For example, if two nodes join a clique of size (k - 1) simultaneously, they will end up in a clique of size (k + 1). Or, if two *swap* operations take place on the same clique simultaneously, then the weight of the clique might end up lower than it was before the operations. Therefore, the existence of a point of synchronization per clique is required. We don't address in this work the situation when leaders may fail.

Like many epidemic peer-to-peer protocols, the PeerMatcher algorithm relies on communication between nodes selected *uniformly at random*. To that end, we rely on the family of peer sampling protocols [14], and specifically Cyclon [11], which provides each node with a regularly refreshed list of links to random other nodes, in a fully decentralized manner and at negligible bandwidth cost. In Cyclon each node maintains a (very short) *partial view* of the network, that is, a handful of links (IP addresses and ports) to other nodes. Each node periodically gossips with one of its neighbors, mixing their views. As a result, views are *periodically refreshed* with new links to random other nodes. This method has shown to produce overlays that strongly resemble random graphs, that is, at any given moment each node's view contains links to nodes selected uniformly at random among all alive nodes [14]. Then, selecting one of these neighbors at random (e.g., to send a pull request), is essentially equivalent to selecting one node at random out of the whole node population. Further, when the node's Cyclon view is changing over time, the node has essentially access to an endless stream of random nodes to communicate with.

Fig. 1 gives an overview of the PeerMatcher architecture. Communication between nodes takes place at two layers. First, a node's PeerMatcher layer talks to another node's corresponding layer to join or swap cliques, as will be explained in the remaining of this section. Here, each node maintains locally a cache of seen cliques (i.e., sets of node IDs that belong to the same clique and their respective addresses), which

[1]we assume that the weights between nodes are static

is updated in a lazy fashion. Second, nodes' peer sampling service layers gossip with each other to maintain a connected overlay and provide a continuous stream of uniformly random nodes picked out of all alive nodes of the network.

B. Phase 1 - Clique Formation

In the first phase of the protocol, nodes connect with their neighbors in order to form cliques of size k.

At the beginning there are no cliques, so nodes connect with their neighbors and form cliques of size 2 (Fig. 2a). In this stage, each node looks in its local *view*, picks the node that is connected to it by the highest weight and sends it a *join* request. Then, the node enters the *waiting state* until it receives a reply. In PeerMatcher, a node being in the *waiting state* rejects all the incoming requests. If the reply does not arrive within a certain timeout, the request is sent again. We assume that the network cannot suffer permanent failures, so it is guaranteed that eventually the request arrives. If the recipient of the request is in the *waiting state* or has grouped with other nodes in the meantime, it replies with a NAK. Otherwise, it sends an acknowledgement (ACK). Since the ACK can also be lost, we use an additional acknowledgement (called ACK2) to make sure that both nodes come to an agreement. At the end of the process, both nodes use the underlying *peer sampling service* to let other nodes know about the newly formed clique. This is done by piggybacking a description of the new clique on the gossiping messages used by the *peer sampling service*.

Now, we are in a situation where some nodes are in cliques of size 2, while others are still alone. If k is 2, then single nodes continue to form pairs following the protocol above until all of them have a match. If k is greater than 2, then each node that is not part of a clique, searches in its local cache for available cliques. It then chooses the one that has the largest weight and sends a *join* request to its leader, which is the node with the highest ID (Fig. 2b). The sender includes in the request a hash of the remote clique's members (i.e., a hash of their IDs) as currently known by the sender. This hash serves as a "signature" of the sender's current knowledge regarding the remote clique's constitution and is guaranteed to be different for different sets of IDs. Upon receiving the request, the leader of the remote clique first checks if the attached hash value is in accordance with the clique's current membership. If this is the case, it means that the operation is valid[2], so it sends an acknowledgement (ACK) to the sender, informing it that it can join the clique. Finally, the latter replies with an ACK2. If the hash value does not match, it means that the sender's view of the remote clique was outdated. The remote clique's leader cancels the request by sending a NAK. Moreover, it includes a fresh listing of the clique members in the NAK message, so the sender node can update its view. This way, we make sure nodes do not send unsuccessful requests over and over, a process known as *starvation*.

[2]The sender must have checked the validity before making the request, which the remote clique leader can trust, as we assume nodes are not malicious and experience no byzantine faults.

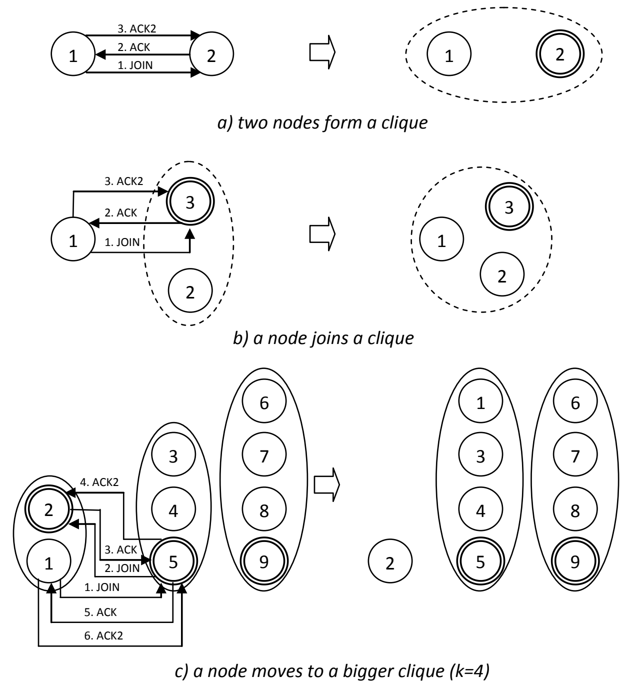


Fig. 2. Phase 1 of PeerMatcher

If we apply only the rules described above in the first phase of the protocol, then we might end up in a situation where all nodes are grouped in cliques, but not all the cliques have k members. For example, if we consider a network with 10 nodes and k is 5, then it is possible to have at the end of the first phase two cliques with three members each and one clique with four members, which is not desirable. To avoid this situation, in PeerMatcher nodes from incomplete cliques constantly try to move to bigger cliques. When a node moves from a smaller clique to a bigger one, we have to make sure that the consistency between the two cliques is maintained (i.e., the operation is perceived as atomic). To this end, we use a synchronization protocol between the node that wants to move and the leaders of the two cliques involved (Fig. 2c). First, the node sends a *join* request to the leader of the clique it wants to move to, including the hashes of *both* its current clique and the remote clique. Upon receiving the request, the leader of the remote clique checks if its clique's current state matches the corresponding hash value in the message. If not, it breaks the operation by sending back a NAK. Else, if the sender's view of the remote clique was up-to-date, it further forwards that *join* request to the leader of the sender's clique and then waits until it receives a response from the latter. The sender's clique leader also verifies (using the other hash value) that no membership change has taken place since the request was issued, and in that case it approves the move: it removes the requester from its clique (this is a local operation for the leader), and sends an ACK to the remote leader. This one sends back an ACK2, then adds the requester in its clique and sends the latter an ACK, letting it join the clique. Finally, the

requester replies with an ACK2. Upon membership change, a leader propagates the updated clique information to all clique members.

If any of the two leaders detects an inconsistency, it cancels the request by sending a NAK which is propagated all the way back to the original requester. Updated clique information is piggybacked in the NAK message, allowing outdated nodes to lazily update their caches on foreign cliques, thus avoiding livelocks. Also, if any of the two leaders is involved in another operation at the time of receiving the request, it rejects the request by sending a NAK. We adopt this policy to avoid deadlocks.

As we assume that links between nodes may suffer transient failures, let us see what happens when different types of messages get lost: (i) if a *join* request gets lost, then it is resent until an ACK or NAK is received back; (ii) if an ACK or NAK gets lost, then the node waiting for it resends the request until it succeeds; (iii) if an ACK2 gets lost, then the node waiting for it resends the ACK until it receives the ACK2. A message is considered lost if a reply is not received within a certain timeout. Our protocol also handles the cases when a message is sent and received twice, due to delays in the network that are longer than the timeout, by assigning timestamps to messages.

We need this synchronization scheme to prevent situations where one of the two cliques changes while the *join* operation is still in progress. By passing the request to the leaders of both cliques, we make sure that no other operation takes place at the same time. In this phase of the protocol, only nodes that are not leaders are allowed to move from one clique to another. This measure helps in keeping the synchronization scheme simple, while not having any noticeable impact on the performance of the algorithm.

C. Phase 2 - Clique Swapping

In the second phase of the protocol, nodes belonging to complete cliques (cliques with k members) continuously swap places in order to improve their cliques. As we will see, there are no interactions between nodes being in this phase of the protocol and nodes that belong to incomplete cliques, as such interactions could lead to inconsistencies between cliques. We also show later that the process of swapping places cannot last forever and the system is guaranteed to converge to a stable state.

Looking for nodes to swap with is an entirely *local* operation performed by nodes operating in the second phase of the protocol. More specifically, to assess the utility of a *swap*, nodes rely on clique configuration updates received *proactively* through the following channels: (i) remote clique information piggybacked on random nodes' Cyclon messages, (ii) updates sent by remote leaders through NAK packets, and (iii) updates regarding the node's own clique propagated by the local leader.

The criteria a node applies to select possible candidate nodes to swap with are based on the clique's weights before and after the *swap*. A *swap* operation is of interest to a node if the weight of its new clique (after the *swap*) will be higher than the weight of its current clique. However, this is not enough.

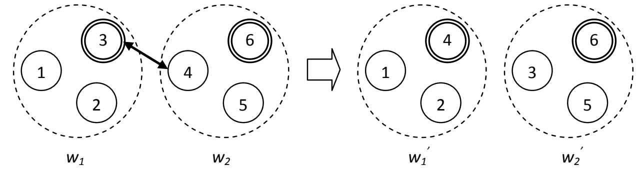


Fig. 3. Swap operation

Additionally, the new weights of both cliques involved in the *swap* should be equal or higher after the *swap* than they were before. Otherwise, both cliques' members have no incentive of consenting to the *swap*, therefore the node has no chances in requesting it.

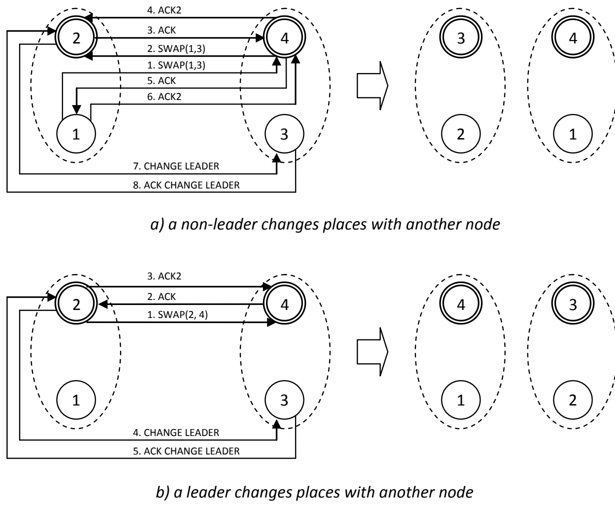
Let us take a look at the example of Fig. 3. It portrays a *swap* between nodes 3 and 4. The operation is considered valid if the following relations hold: $w_2'>w_1$, $w_2'\geq w_2$ and $w_1'\geq w_1$. As explained above, the first relation determines whether the *swap* is beneficial for node 3, while the second and third ensure that it will be accepted by the remaining members. We can also deduce from the relations above that the weight of a clique can only increase in time, which, combined with the assumption that the network is finite, leads us to the conclusion that the weights of all cliques cannot grow indefinitely, so the system allways converges to a stable state. Notice that we do not need to add $w_1'\geq w_2$ as an extra condition because we consider only the case when nodes are selfish, so node 3 does not care if node 4 ends up in a clique that is worse that its previous one.

As *swap* operations modify the membership of cliques, they have to be performed atomically with respect to each other. That is, we have to make sure that no other *swap* operation that alters one of the two cliques involved is performed at the same time. To achieve this, we put in place a synchronization mechanism similar to the one we use for nodes that move between cliques in the first phase of the protocol.

When a node changes places with another node, we have to follow a set of steps to make sure that no other change is made to the cliques of the two nodes at the same time. These steps are very similar to the ones taken when a node moves to a bigger clique in the first phase of the protocol. Like there, when a node finds that by changing places with a neighbor from a different clique the weight of its clique can improve, it first sends a request to that neighbor's clique leader. Then, the leaders of the two cliques check the hash values in that message to verify that the consistency is not broken by performing the *swap* and, if both of them agree, the operation takes place. Otherwise, the operation is cancelled and the states of the cliques remain unaltered.

Unlike the first phase of PeerMatcher now we allow both leaders and non-leaders to swap cliques. This creates two possible scenarios. In the first case, the node that issues the request is a non-leader (Fig. 4a), while in the second case it is a leader (Fig. 4b).

It may happen that nodes that are in the first phase of the protocol send erroneous *join* requests to nodes that are in



a) a non-leader changes places with another node

b) a leader changes places with another node

Fig. 4. Phase 2 of PeerMatcher

```
1:  function ACTIVE_THREAD()
2:      loop
3:          importCliquesFromCyclonNeighbors()
4:          if isNodeWithoutClique() then
5:              node←selectNeighborToFormClique()
6:              sendReqToJoinClique(node)
7:              waitingState←true
8:          else if isNodeInPartialClique() then
9:              node←selectLeaderOfBiggerClique()
10:             sendReqToJoinClique(node)
11:             waitingState←true
12:         else if isNodeInCompleteClique() then
13:             node←selectNeighborToSwapClique()
14:             sendReqToSwapClique(node)
15:             waitingState←true
16:         end if
17:     end loop
18: end function
```

Fig. 5. The PeerMatcher active thread.

the second phase because they have an outdated *view* of the system. Nodes that receive such requests always reply with a NAK in which they include the current state of their clique, so the requester can update its *view*. This way we prevent potential inconsistencies and also livelocks.

D. Implementation Details

We implemented PeerMatcher on top of PeerSim [15], an open source framework that allows simulation of peer-to-peer protocols in an event-driven environment. One big advantage of PeerSim is that it allows stacking multiple peer-to-peer protocols on top of each other, a feature that fits perfectly our system architecture from Fig. 1.

We implemented PeerMatcher as two threads, an active one and a passive one. The *active thread* is executed periodically, to check and update the local state and to initiate gossip-based communication. The *passive thread* receives messages asynchronously, processes them, updates the local state, and if needed sends a response or other messages. Figures 5 and 6 present the pseudocode for our implementations of the active and passive thread, respectively.

In the first line of the active thread, the node retrieves an updated list of neighbors from Cyclon. Each entry contains the node's ID, the node's address, and the node's clique (i.e., the IDs and addresses of nodes in that clique, but *not* their cliques recursively). This list of fresh neighbors is used in the subsequent *select** functions as a pool of nodes to select from for forming a clique or for exploring swap opportunities.

Subsequently, a node performs one of the following three operations, depending of its state, after which it enters the *waiting state*:

- If the node is not a member of any clique, it searches for a neighbor in order to form a clique with it or to join its clique, if the neighbor is already in a clique.
- If the node takes part in a partial clique (a clique that has less than k members), it searches for a bigger partial clique to join it. By doing so, we prevent the protocol from getting stuck in a configuration in which not all the cliques are complete.
- If the node is in a complete clique, it searches for a neighbor to swap positions with, if this can lead to a new clique of higher weight.

Fig. 6 depicts the pseudocode of the PeerMatcher's passive thread. This code is executed every time a node receives a message from another node. There are six types of messages a node can receive, which are handled as follows:

- When a node receives a *request to join clique*, we identify two possible cases: (i) if the node is alone, then it adds the sender of the message in its clique and responds with an ACK; (ii) if the node belongs to a clique and the *join* operation is valid (it does not break the consistency), an ACK is sent to the sender if the sender is a clique leader, otherwise the request is forwarded to the sender's clique leader. If the operation is not valid, a NAK is sent to the sender.
- Upon receiving a *request to swap cliques*, the same actions are taken as if the node had received a *request to join clique*.
- When an *ACK* is received, the node applies the corresponding modifications to its clique and, if it is not last in the ACK chain (e.g. node 4 in Fig. 4a), it sends the ACK to the issuer of the request.
- An *ACK2* means that the *join* or *swap* operation completed successfully and the node can safely exit the *waiting state*.
- If a *broadcast message* from the local leader is received, the node updates the information it has regarding the members of the clique it belongs to.
- Receiving a NAK means that the node has an outdated view of the target clique from its original request, so it updates its *view* of that clique with the information provided in the NAK message.

At the end of the passive thread's execution, if the node is

```
1: function PASSIVE_THREAD
2:     loop
3:         event←receiveFromAny()
4:         if waitingState=true then
5:             sendNak(event.sender)
6:             exit()
7:         end if
8:         if isReqToJoinClique(event) then
9:             if isNodeWithoutClique() then
10:                updateClique(event)
11:                sendACK(event.sender)
12:                waitingState←true
13:            else
14:                if isValidReq(event) then
15:                    if hasToForwardReq(event) then
16:                        forwardReq(event.remoteLeader)
17:                        waitingState←true
18:                    else
19:                        updateClique(event)
20:                        sendACK(event.sender)
21:                        waitingState←true
22:                    end if
23:                else
24:                    sendNAK(event.sender)
25:                end if
26:            end if
27:        else if isReqToSwapClique(event) then
28:            if isValidReq(event) then
29:                if hasToForwardReq(event) then
30:                    forwardReq(event.remoteLeader)
31:                    waitingState←true
32:                else
33:                    updateClique(event)
34:                    sendACK(event.sender)
35:                    waitingState←true
36:                end if
37:            else
38:                sendNAK(event.sender)
39:            end if
40:        else if isACK(event) then
41:            updateClique(event)
42:            if hasToForwardACK(event) then
43:                forwardACK(event.issuer)
44:            else
45:                waitingState←false
46:            end if
47:            sendACK2(event.sender)
48:        else if isACK2(event) then
49:            waitingState←false
50:        else if isNAK(event) then
51:            updateClique(event)
52:            waitingState←false
53:        else if isBroadcast(event) then
54:            updateLocalClique(event)
55:        end if
56:        if isLeader() and myCliqueWasUpdated() then
57:            broadcastUpdateToMyPartners()
58:        end if
59:    end loop
60: end function
```

Fig. 6. The PeerMatcher passive thread.

the leader of its clique and the clique has been updated as a consequence of the received event, it broadcasts the updated clique's configuration to all members of the clique.

To keep matters simple, in the pseudocode from Fig. 6 we omitted the case when a node transfers leadership to another node after a *swap* operation (see Fig. 4). This transfer of leadership takes place inside the *updateClique(event)* method if the current node loses the leadership of the clique as a result of having a new member with a higher ID in the clique. The leadership is transferred by sending a *leadership change message* to the new member, which has to acknowledge the operation with an ACK message.

V. Evaluation

A. System Setup

In this section we examine the efficiency of PeerMatcher in clustering nodes of complete weighted graphs into cliques of a fixed size k. In our measurements, we are interested in the amount of computation and number of rounds needed by our protocol to find valid and stable matchings and also in the *weight* of the matchings found. We compute the *weight* of a matching as the arithmetic mean of the weights of its cliques. In turn, the weight of a clique is computed as the arithmetic mean of the edges' weights. We also tested the geometric mean as an alternative, but did not notice any performance implications.

We benchmarked our protocol against the one described in [9], which is an improved version of the one from [2]. To our knowledge, this is the only alternative for computing the maximal *k-clique matching* of a graph in a distributed manner. For readability convenience we will be referring to the protocol proposed in [9] as *CliqueFinder* for the remaining of the paper. In our experiments, we used the *Pruning* version of CliqueFinder that uses partial views, as this one gave the best results. We could not compare against the optimal clique matching as finding the latter is computationally infeasible. Also, to our knowledge there is no efficient centralized algorithm that solves the k-clique matching problem to compare against.

As input for each simulation, we used a list of weights between all pairs of nodes, whose values are uniformly distributed in the interval (0,1). This list of weights is only used by nodes to assess the benefit of their partnership with other nodes and does not reflect the knowledge they have about the network. For network discovery, nodes use the peer sampling service we discussed in Section IV. It would be interesting to see how the protocol performs for input sets of weights generated following a distribution other than uniform random, as this might influence the network traffic between the nodes. However, our measurements showed that the bandwidth usage of our protocol is very low, which motivated us to leave this topic for future research.

All simulations were performed in PeerSim [15]. Each of the plots presented here was obtained by averaging the results of 5 independent simulations.

B. Performance

The performance of PeerMatcher (or any other distributed protocol that aims to solve the *k-clique matching problem*)

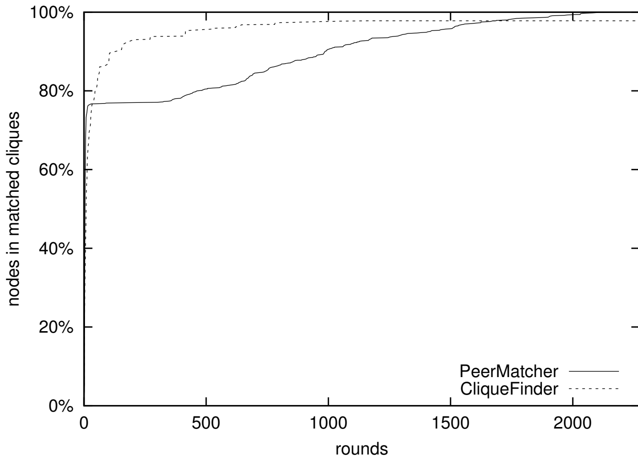


Fig. 7. Convergence for PeerMatcher and CliqueFinder, as a function of the number of rounds elapsed.

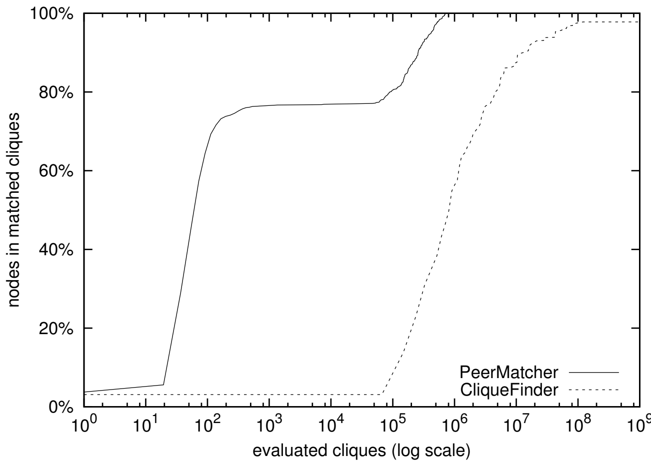


Fig. 8. Convergence for PeerMatcher and CliqueFinder, as a function of the number of evaluated cliques.

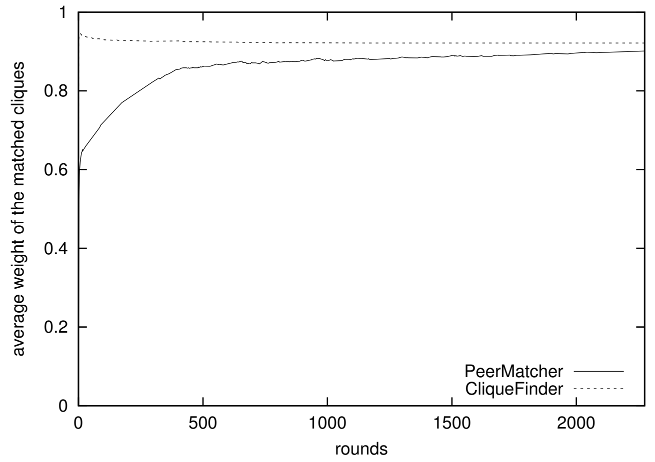


Fig. 9. Average weight of the matchings found by PeerMatcher and CliqueFinder, as a function of the number of rounds elapsed.

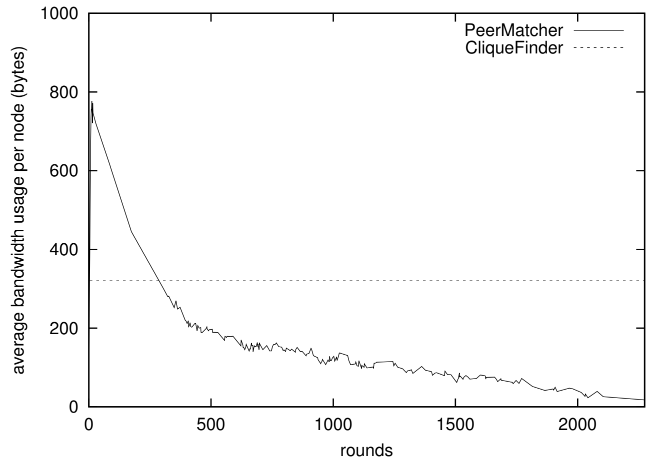


Fig. 10. Communication footprint for PeerMatcher and CliqueFinder, expressed as the number of bytes sent per round.

depends on three major factors: (i) the number of rounds needed to converge, (ii) the amount of computation required, and (iii) the weight of the matching found. To assess these attributes we compare PeerMatcher against CliqueFinder in a network of 500 nodes that we aim to partition into cliques of size 5.

Note that, unlike PeerMatcher that uses the clique membership protocol we discussed in Section IV to ensure that cliques do not overlap, in CliqueFinder there is no such mechanism. Instead, each node maintains its own independent state about the clique it believes it belongs to, without knowing whether the remaining clique members share the same clique state. Although all individual views are eventually aligned in the converged state, this is not necessarily the case during the operation of the protocol. As such, CliqueFinder [2] introduces the notion of a *matched* clique. A clique is *matched* at a given moment in time if it has k members and all of them share the same perception of the clique. Given this definition, there are two performance aspects that matter most at a given point in time during the execution of the CliqueFinder protocol: the percentage of nodes in matched cliques and the average weight of the matched cliques.

In Fig. 7 the fraction of nodes that belong to matched cliques is depicted as a function of the number of rounds elapsed. The first thing we notice here is that CliqueFinder has a better start, having 95% of nodes grouped into matched cliques after 500 rounds, while PeerMatcher has only 80%. However, after 2000 rounds all nodes have a clique in PeerMatcher, while in CliqueFinder there are 15 nodes left (3% of the total) searching for a clique until the experiment ends. This results from the fact that in CliqueFinder nodes blindly search for the best clique among all nodes, without prioritizing nodes that don't have a clique. The last thing to note here is that PeerMatcher has a steep convergence rate until reaching 70% (in round 7), after which it gradually converges to 100%. The reason is that in round 7 a large fraction of nodes passes from phase 1 of

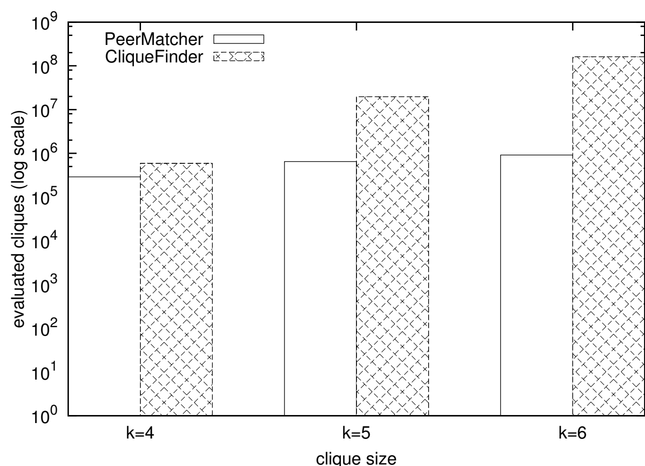


Fig. 11. Scalability with respect to the clique size.

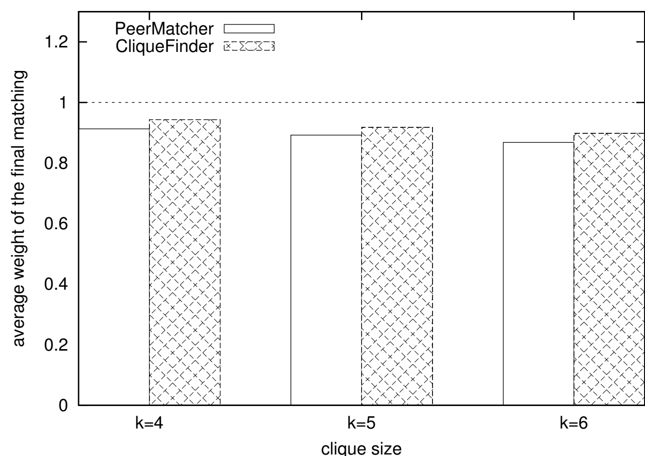


Fig. 12. Average weight of the final matching for different clique sizes.

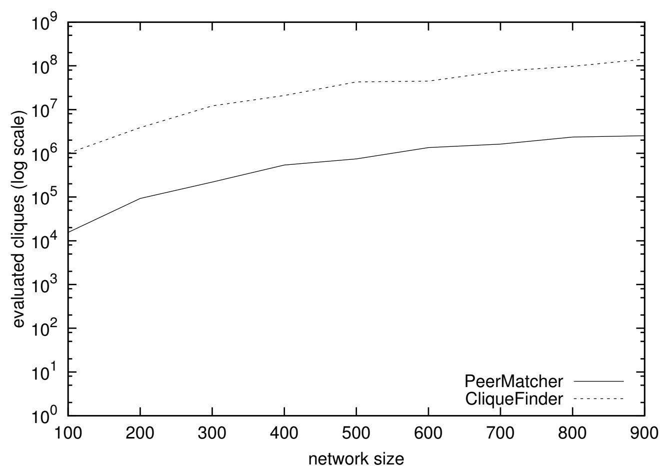


Fig. 13. Scalability with respect to the network size.

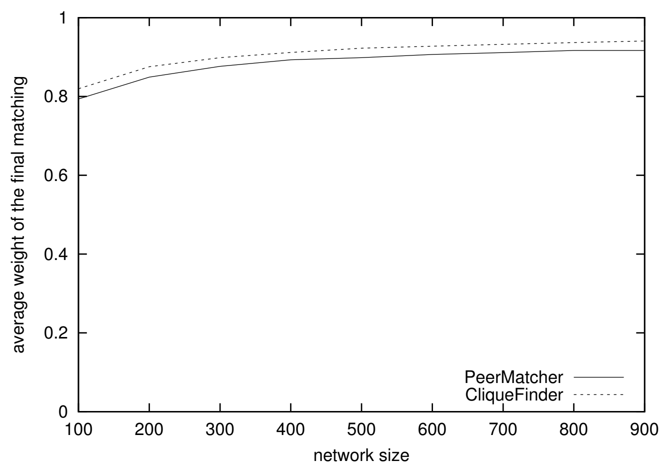


Fig. 14. Average weight of the final matching for different network sizes.

the protocol to phase 2, thus determining an abrupt increase in the number of matched cliques.

A completely different picture arises when looking at Fig. 8, which shows the number of nodes in matched cliques as a function of the average number of cliques evaluated by a node. This metric indicates the computational complexity of the algorithms. We can see here that PeerMatcher converges more than 10 times faster when compared to CliqueFinder. This difference reflects the smaller number of operations performed in each round by PeerMatcher. Recall that in each round, a PeerMatcher node only iterates over its neighbors' cliques and picks the best neighbor for a *join* / *swap* operation. A CliqueFinder node, however, evaluates in each round a number of cliques, and this number grows exponentially with the size of the clique. This was confirmed by our measurements, which showed that PeerMatcher nodes evaluated 400 cliques per round on average in this experiment, while CliqueFinder nodes performed around 100,000 evaluations per round. When looking at the PeerMatcher curve from Fig. 8, we notice that there is a period of stagnation in the middle. This can be explained by the fact that during this period most of the nodes are in the second phase of the protocol, so a large number of cliques are evaluated for potential swaps, but only few new cliques are formed.

Fig. 9 shows that both protocols find quality matchings, with weights above 0.9. As mentioned earlier, we cannot compare our results against the optimal matching of the given input graph, due to the high cost of computing the latter. However, we know that the weight of such optimal matching cannot exceed value 1, since the weights of the edges are uniformly distributed between 0 and 1. Therefore, we can safely say that the matchings found by both protocols are at most 10% off from the optimal one. When comparing the two protocols with each other, we see that CliqueFinder performs slightly better for the whole run. Yet, the difference between them is marginal, varying from 6% in round 500 to only 2% at the end of the simulation. This is normal, given the fact that in CliqueFiner nodes evaluate 250 more cliques in each round, thus having more options to choose from.

The descending curve produced by CliqueFinder in Fig. 9

might give the impression that CliqueFinder finds an excellent matching after just a couple of rounds, which slightly decreases in quality after that. What happens in reality is that after 6 rounds CliqueFinder finds only 192 matched cliques (38.4% out of total), whose average weight is indeed very good (0.94), while PeerMatcher finds 322 matched cliques with an average weight of 0.60, which later on goes up to 0.90. This observation attests the fact that PeerMatcher puts the accent on convergence speed at the beginning, by allowing nodes to rapidly group into k-cliques in its first phase, while in CliqueFinder nodes are more preoccupied with finding good cliques, at the expense of slower convergence rates.

Another factor that impacts the performance of a peer-to-peer protocol is the network traffic generated by peers. In the next experiment we measure the average amount of network traffic processed by each peer in a round (Fig. 10). After 2000 rounds, a PeerMatcher node experienced an average total traffic of 360 KB, while for CliqueFinder the average total traffic per node was 720 KB. If we choose a round duration of 1s, we obtain a bandwidth usage of 1.44 kbps for PeerMatcher and 2.88 kbps for CliqueFinder. Again, PeerMatcher outperforms CliqueFinder by a factor of 2. This happens because in CliqueFinder each node sends its clique at the end of each round to all clique members, while in PeerMatcher the bandwidth usage gradually decreases over time, as an effect of the number of clique changes that occur, which gets smaller with each round. For this experiment, one could argue that if we stopped CliqueFinder earlier, then the bandwidth usage would become zero from that point on. However, if we did that, then the matching found by CliqueFinder would be incomplete, as CliqueFinder requires the whole timespan shown on the x axis to converge to a valid matching.

C. Scalability

It is interesting to see how PeerMatcher performs for different setup configurations. To achieve this, we performed a number of simulations for different clique and network sizes.

First, we measured the amount of computation required for cliques of size 4, 5 and 6 (Fig. 11), while keeping the network size fixed to 500. We can observe that the number of evaluated cliques in PeerMatcher hardly depends on the clique size, while in CliqueFinder it grows exponentially with it (notice the logarithmic scale on the vertical axis). Things become even more interesting if we look at Fig. 12, which shows that both protocols produce final matchings that are almost equally good, with CliqueFinder outperforming PeerMatcher by only 3% for cliques of size 6.

Second, we tested for network sizes between 100 and 900 (Fig. 13), while keeping the clique size fixed to 5. Again, we see that there is a difference of almost two orders of magnitude between the two protocols. Things become even better when looking at Fig. 14, which indicates that the final matchings found by our protocol are less than 2% off by those found by CliqueFinder for the whole range of network sizes. This is a strong indicator that PeerMatcher can be used in large networks where a high convergence speed is needed.

VI. Conclusions

In this paper, we proposed and evaluated a new distributed protocol that approximates the solution of the k-clique matching problem. Unlike previous approaches, our protocol converges faster and maintains the consistency of the system during its execution, while still providing quality matchings. The evaluations we conducted indicate that our protocol performs well in large networks and for different clique sizes.

We believe that our research represents an important step towards the goal of designing a reliable, general-purpose distributed system for solving large-scale matching problems from fields like marketing or social media.

Acknowledgments

This publication was supported by the Dutch national program COMMIT/. We thank Anna Chmielowiec for providing us with the code of CliqueFinder and also for her comments that greatly improved the manuscript. We are also grateful to the four anonymous reviewers for their useful insights.

References

[1] K. Ashton, “That ‘Internet of Things’ thing,” *RFiD Journal*, vol. 22, pp. 97–114, 2009.

[2] A. Chmielowiec and M. van Steen, “Optimal decentralized formation of k-member partnerships,” in *Self-Adaptive and Self-Organizing Systems (SASO), 2010 4th IEEE International Conference on*. IEEE, 2010, pp. 154–163.

[3] H. N. Gabow, “Data structures for weighted matching and nearest common ancestors with linking,” in *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 1990, pp. 434–443.

[4] D. G. Kirkpatrick and P. Hell, “On the completeness of a generalized matching problem,” in *Proceedings of the tenth annual ACM symposium on Theory of computing*. ACM, 1978, pp. 240–245.

[5] S. Micali and V. V. Vazirani, “An $o(v|v|c|e|)$ algorithm for finding maximum matching in general graphs,” in *Foundations of Computer Science, 1980., 21st Annual Symposium on*. IEEE, 1980, pp. 17–27.

[6] H. N. Gabow and R. E. Tarjan, “Faster scaling algorithms for general graph matching problems,” *Journal of the ACM (JACM)*, vol. 38, no. 4, pp. 815–853, 1991.

[7] V. Kann, “Maximum bounded h-matching is max snp-complete,” *Information Processing Letters*, vol. 49, no. 6, pp. 309–318, 1994.

[8] F. Manne and M. Mjelde, “A self-stabilizing weighted matching algorithm,” in *Stabilization, Safety, and Security of Distributed Systems*. Springer, 2007, pp. 383–393.

[9] A. Chmielowiec, S. Voulgaris, and M. van Steen, “Decentralized group formation,” *Journal of Internet Services and Applications*, vol. 5, no. 1, pp. 1–18, 2014.

[10] F. Rahimian, A. H. Payberah, S. Girdzijauskas, M. Jelasity, and S. Haridi, “Ja-be-ja: A distributed algorithm for balanced graph partitioning,” in *Self-Adaptive and Self-Organizing Systems (SASO), 2013 IEEE 7th International Conference on*. IEEE, 2013, pp. 51–60.

[11] S. Voulgaris, D. Gavidia, and M. Van Steen, “Cyclon: Inexpensive membership management for unstructured p2p overlays,” *Journal of Network and Systems Management*, vol. 13, no. 2, pp. 197–217, 2005.

[12] M. Jelasity, W. Kowalczyk, and M. Van Steen, “Newscast computing,” Technical Report IR-CS-006, Vrije Universiteit Amsterdam, Department of Computer Science, Amsterdam, The Netherlands, Tech. Rep., 2003.

[13] E.-G. Talbi, *Metaheuristics: from design to implementation*. John Wiley & Sons, 2009, vol. 74.

[14] M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. van Steen, “Gossip-based Peer Sampling,” *ACM Transactions on Computer Systems*, vol. 25, no. 3, p. 8, Aug 2007.

[15] A. Montresor and M. Jelasity, “Peersim: A scalable p2p simulator,” in *Peer-to-Peer Computing, 2009. P2P’09. IEEE Ninth International Conference on*. IEEE, 2009, pp. 99–100.