RankSlicing: A decentralized protocol for supernode selection

Giovanni Simoni
Peerialism AB, and
KTH - Royal Institute of Technology
giovanni.simoni@peerialism.com

Roberto Roverso
Peerialism AB
roberto.roverso@peerialism.com

Alberto Montresor
University of Trento
alberto.montresor@unitn.it

***Abstract*—In peer-to-peer applications deployed on the Internet, it is common to assign greater responsibility to supernodes, which are usually peers with high computational power, large amount of memory, or high network bandwidth capacity. In this paper, we describe a practical solution to the problem of supernode selection, that is the process of discovering the best peers in the network by some application-specific metric. We provide a distributed heuristic that allows to identify the best K nodes in the P2P overlay, by taking into consideration the realities of actual deployments, such as the presence of NATs. Our approach consists of an epidemic protocol which does not require new connections to be established, but rather relies on established connections, such as the ones provided by a NAT-resilient peer sampling framework. We support our claims with a thorough evaluation of our solution in simulation and in a real deployment on thousands of consumer machines.**

I. Introduction

In peer-to-peer applications deployed on the Internet, it is common to assign greater responsibility to the "best" peers, where the notion of "best" is based on metrics such as computational power, amount of memory and network bandwidth capacity. Such peers are usually referred as *supernodes*. In distributed storage applications, for instance, peers with large persistent memory and longer lifetime are likely to become supernodes and therefore host replicas of content [14]. Supernodes are also used to work around connectivity issues [12] by either relaying NAT traversal messages or even actual content when no direct connectivity can be established [1]. In P2P live streaming, supernodes with high upload capacity are tasked to retrieve the streaming data directly from the source of the stream and distribute it to a large number of neighbors [13]

In this work, we tackle the *supernode selection* problem, that is the decision of which peers, among all nodes in the overlay, should become supernodes. This effort is motivated by the needs of a commercial peer-assisted live streaming platform called Hive Streaming [19][1], which utilizes supernodes to efficiently deliver content to the viewers. Peer-assisted streaming applications strive to provide the same quality of user experience as CDNs in terms of throughput and latency, while keeping the load on the source of the stream to a minimum. There are two main requirements for supernode selection. The first is that the supernodes must be the peers with the highest upload capacity. It has been shown that this can significantly decrease the average number of hops the content has to traverse and therefore also lower latency, significantly improving the quality of user experience [3]. The second requirement is that the size of the supernode set has to be kept – as much as possible – fixed over time and equal to a design parameter K. The value K is a system parameter that depends on the particular application; e.g., in P2P live streaming applications [5] [10], K can be derived from the number of peers and their upload bandwidth distribution. In such case, the rationale behind limiting the amount of supernodes is to provide bounds on the load and distribution costs of the streaming source, while keeping a good level of the quality of user experience.

Supernode selection can be simply achieved by having nodes promote themselves as supernodes if their resources are above a certain threshold [17] [1]. With this type of solution, however, the supernode set can become arbitrarily large and change drastically in size because of churn. A better alternative for supernode selection is to use a distributed slicing algorithm [4] that allows the identification of a *slice*, i.e. a group of nodes in the overlay that again meet a minimum threshold on available resources. Nodes in a slice can then be promoted to supernodes by the application.

In the literature, two types of distributed slicing algorithms can be found: *absolute slicing* [11] and *ordered slicing* [6]. However, none of the two approaches address our requirements of finding the best K peers in the overlay network, but rather either choose any K peers among the potential supernodes (absolute slicing) or the best X percentage of nodes among all nodes in the system (ordered slicing).

Another issue with the state of the art is that all distributed slicing algorithms assume that peers can at all times establish direct connections between themselves because they are based on gossip. This is a rather unrealistic assumption in real deployments where most of the peers are behind network address translator (NAT) and cannot achieve direct connectivity, even using NAT traversal mechanisms [8].

The contribution of this paper is a practical solution to the problem of supernode selection called RankSlicing. Beside allowing the identification of the best K nodes in the system under realistic deployments characterized by the presence of NATs, RankSlicing has been designed while keeping the following additional set of informal requirements in mind:

- The stability of the supernode set is of paramount importance. The motivation for providing a supernode selection algorithm, in the first place, is that the application needs to delegate a certain role to the supernodes. Frequent and/or

[1] www.hivestreaming.com

978-1-4799-6201-3/14/$31.00 2014 IEEE

abrupt changes of the supernode set tend to disrupt the application. Therefore, it is important that the supernode set remains stable over time if no node better than any of the existing supernodes joins the network or one of the current supernodes leaves the overlay.
- All nodes should be aware of which peer nodes belong to the supernode set; e.g., in live streaming, this is to easily locate supernodes and request content from them.
- Each node should have an estimate about the stability of its supernode set. This is to avoid executing expensive operations if a supernode set if not stable.

RANKSLICING is based on an epidemic protocol for supernode selection, where each peer maintains a view of the supernode set and keeps that up-to-date over time by continuously exchanging information with its neighbors. Our approach is NAT-aware, that is it does not require the establishment of new connections but rather rely on a NAT-aware topology management system such as the one provided by NAT-resilient peer sampling protocols, such as WPSS [18] or Croupier [2]. We evaluate our approach both in simulation and in a real network on thousands of consumer machines. In our experiments, we show that our approach is highly resilient to churn and that meets all of the requirements described above.

The rest of the paper is organized as follows. Section II introduces the work related to distributed slicing. Section III describes the details of the problem to be solved, while Section IV discusses our solution. Experimental results are shown in Section V, while the paper is concluded by Section VI.

II. RELATED WORK

Absolute slicing, mentioned in the introduction, constitutes in our opinion the most closely related work to RANKSLICING since it aims at identifying a slice of the overlay network of size K. Absolute slicing achieves supernode selection using a three-layered overlay network. The outermost layer is a random overlay that contains all peers and is built using Newscast [7]. The second overlay is constructed in the same way but only eligible nodes are allowed to participate. Eligibility is determined by a threshold on an application-specific metric. The last layer, called *slice*, is composed of K peers on average and it is constructed by having peers probabilistically decide if to join the layer according to the number of peers already in the layer and the value of K. The approach is particularly suited for churn as it is based on gossip and it can maintain an average number of nodes in the third layer that is equal to K over time. However, the set of nodes in the slice changes continuously even without churn, which is unreasonable for our requirement of stability of the supernode set. Besides that, absolute slicing does not promote the best K peers in the overlay to the slice layer but rather any K number of the eligible nodes.

Sacha et al. [20] propose a distributed ranking method called *Gradient*, similar to [7]. The approach is based on having nodes periodically measure their utility value through a utility metric and gossip about the measured value with their neighbors. Each peer then maintains a set of similar neighbors, with respect to the utility value, along with a set of randomly selected ones. The main application of the distributed ranking in this case is search. Peers can issue queries for identifying which peers have higher or same rank as specified in the query. This can be used to find supernodes with high utility values, however queries cannot be instrumented to return the top-K peers according to those values.

Raychoudhury et al. [16] propose a strategy for identifying the top K nodes in the context of wireless ad-hoc networks. Each peer locally finds the directly reachable nodes and then all peers proceed with the election of a leader supernode. The leader supernode later in turn selects the remaining $K-1$ super-peers. The protocol however does not tackle the problem of continuously adapting the supernode set according to the changes in the overlay network. The only way to address churn is for the application to expressly restart the supernode selection process.

A recent paper from Liu et al. [9] proposes an algorithm for supernode selection based on gossip. Nodes start as regular peers, to later build the set of candidate supernodes by evaluating the utility values of their neighbors. After that, peers make a local decision, based on an application-defined threshold on available resources, if to act as superpeers or not. Although churn is expressly addressed as a problem, the number of selected supernodes in the system cannot be specified by the application in the proposed solution.

Finally, supernode selection can be achieved with structured overlay networks rather than gossip. The SPiDeR [21] framework implements Web discovery services through a structured overlay in which only the most powerful nodes join a structured overlay network based on Chord [22]. In this case, peers become supernodes if their identifiers occupy specific positions in the ring. The same supernode selection technique is used by a distributed filesystem designed by Kovendhan el al. [14]. In our case, we concentrate on an unstructured overlay approach given that it provides better resiliency to churn compared to structured overlay approaches [15].

III. PROBLEM DEFINITION

In this paper, we aim at solving the problem of choosing a supernode set that includes the best nodes in the overlay based on a utility function provided by the application and has fixed size K. As in absolute slicing, we limit the number of peers that can become supernodes by setting a threshold on the utility values of peers. Nodes having an utility value greater than the threshold are considered *eligible* to become supernodes. This is in line with our application of the algorithm, that is as part of a live streaming system, where we set a threshold on, for instance, computation power in order to avoid weak peers to serve as supernodes. We design our approach considering that the view of the supernode set at a peer should adapt as quickly as possible to changes in the overlay, namely churn, such that peers have, at all times, the best view of the top K peers. Due to dynamism in the network and latency however, the view of the supernode set at each peer might slightly differ. We have designed our live streaming application to take into account this limitation. However, the performance of the system as a whole is directly proportional to how correct is the supernode set estimation at peers.

We now define our problem in a more formal manner.

Let $\Pi^t = \{ p_1, \ldots, p_n \}$ be the set of nodes in the network at any given time t. Each node p_i is characterized by a time-varying tuple $C^t(p_i) \in \mathbb{R}^m$ that is our utility value. $C^t(p_i)$

contains m numerical values that measure the capabilities of p_i, in terms of software and hardware resources, at time t. A node can obtain an up-to-date version of such tuple at any time, by calling the *capability function* $\mathsf{cf}()$. The *eligibility predicate* $\mathsf{ep} : \mathbb{R}^m \rightarrow \mathsf{Boolean}$ returns true if a given set of capabilities is sufficient to become a supernode. A node p_i is eligible at time t if $\mathsf{ep}(C^t(p_i))$ is true.

Both the values that are reported by the capability function and the definition of the eligibility predicate are application-dependent. For example, nodes may be associated with a triple $\langle x_1, x_2, x_3 \rangle$, identifying machines with x_1 CPUs, x_2 GB of available storage space and x_3 KB/s of available bandwidth. We could for instance consider eligible any node having at least 16 GB of available storage and 2 CPUs.

We denote the set of eligible nodes at time t as $\mathcal{E}^t \subseteq \Pi^t$. Both the set of nodes and the set of eligible nodes may vary over time: Π^t may differ from Π^{t+1} because of *churn*, i.e. nodes joining and/or leaving the system at any time. $\mathcal{E}^t$, besides being affected by churn, may also change because of variations in the system state: e.g., in the case of storage media, eligibility may change as storage is allocated or deallocated.

We do not consider byzantine failures related to neither nodes nor communication: security aspects are assumed to be covered transparently by the framework underneath.

Under the aforementioned assumptions, the desired output at any node p_i at time t is the *supernode set* L_i^t such that $|L_i^t| = \min\{K, |\mathcal{E}^t|\}$. We call such set the *supernode set* of p_i at time t. Ideally, such sets should be characterized by the following properties:

- **Consistency**: If no variations occur in the eligible set for a sufficiently long time, the supernode sets of any pair of node must eventually converge to the same set.

$$\exists t_0, \forall t \geq t_0 : \mathcal{E}^t = \mathcal{E}^{t+1} \Rightarrow$$
$$\exists t_1 \geq t_0, \forall t \geq t_1, \forall p, q \in \Pi^t : L_p^t = L_q^t$$

- **Adaptiveness**: If no variations occur in the eligible set for a sufficiently long time, each of the supernode sets must eventually be contained in $\mathcal{E}^t$. In other words, nodes that lose their eligible status must eventually leave the supernodes set.

$$\exists t_0, \forall t \geq t_0 : \mathcal{E}^t = \mathcal{E}^{t+1} \Rightarrow$$
$$\exists t_1 \geq t_0, \forall t \geq t_1, \forall p \in \Pi^t : L_p^t \subseteq \mathcal{E}^t$$

IV. Algorithm

The design of RankSlicing is inspired by the following observation: given that the eligibility and selection as supernode of a node depends strictly on its capabilities, such values can be used to reduce supernode selection to a *ranking* problem. So, we first rank eligible nodes based on their utility value and solving the tie-breaks, if any; second, we select the first K items in such ranking to become supernodes.

In our approach, we assume that nodes are already connected through a random overlay topology provided by another service, such as a NAT-resilient peer sampling service. Besides that, in our protocol we strictly avoid creating new connections in order to prevent running into connectivity limitations caused by the presence of NATs. That is because, in gossip algorithms, those limitations have been shown to cause significant biasing towards nodes that are easily reachable [8].

For our implementation, we choose WPSS [18] to provide a NAT-resilient overlay. In WPSS, every peer maintains a slowly changing and fixed set of connections with its neighbors. NATed nodes are connected only to public nodes and public nodes are connected among themselves. Using this type of overlay has a number of advantages. First, the system does not incur in the cost of executing expensive NAT-to-NAT connection establishments. Second, the constructed overlay is random and is not subjected to biasing introduced by the presence of NATs. Finally, WPSS is resilient to churn, that means that failed overlay connections are replaced with new ones such that every peer maintains a constant number of connections to neighbours. Layering our protocol over WPSS allows us to design a protocol that is oblivious to connectivity limitations caused by firewalls and NATs and where churn-related overlay maintenance is greatly simplified.

A. Gossip-based algorithm

The distributed selection of the supernodes set is achieved through a push-pull gossip protocol.

Each node p_i maintains a set called *view*, which is the local approximation of the supernode set, containing up to K *node descriptors*. Each descriptor $\langle i, lc, age, cap \rangle$ is associated to the identifier i of the node p_i that created it, and contains a monotonically increasing logical clock lc, an age parameter age, and the evaluation of node capabilities cap as reported by p_i itself when the descriptor has been created. Two descriptors contained in a view cannot be associated to the same process.

Periodically, each node p_i obtains the identifier of a random peer p_j from the underlying peer sampling protocol and sends a random sample S of the local view to it, adding a freshly emitted process descriptor for itself p_i.

Upon reception of the sample S, the node p_j merges it together with the local view as follows: 1) A merge list M is built by concatenating S and the view V_j; 2) Duplicated descriptors (i.e., descriptors referring to the same process) are removed, by keeping the one with the freshest logical clock. 3) The merge list M is sorted in descending order of capabilities, according to a pre-defined total ordering; 4) M gets truncated to the first K elements; 5) The local view of j gets replaced with the set of descriptors in M.

It should be noticed that this is *not* a peer sampling protocol: the fact that a descriptor of p_k is contained in the view of p_i does not imply a connected neighbourhood relation between p_i and p_k. Consequently, each process is allowed to know the identity of the K supernodes, without creating a new topology (as in absolute slicing [11]) and without affecting the existing one.

On every gossip exchange, the duplication removal phase discards superseded descriptors, that is those that are older than the ones which have been received in the last exchange (for the same nodes). As a result of the sorting and truncation phases, all but the best K descriptors get discarded, while the descriptors of the strongest nodes keep being propagated in the system. In the absence of churn, the eligible set becomes static and all the nodes will eventually share the same set of descriptors, thus satisfying the **Consistency** requirement.

B. Adaptiveness and churn handling

The effects of churn on the overlay structure are assumed to be handled by the underlying topology management service. However, we still need to remove faulty nodes from the supernode set, as they are no longer available for providing their services. Descriptors referring to faulty nodes must be removed from the views of every process. In the same way, we need to address fluctuations of the supernodes set due to evolutions in the node capabilities.

The descriptor field age is used to purge stale pieces of information from views. Intuitively, this field reports the cumulative time spent by a descriptor inside the view of each process. When a fresh descriptor $\langle p_i, lc, age, cap \rangle$ is emitted by process p_i, its age field is 0. As the process p_j accepts a descriptor into its view, a time-stamp t_a is associated to it. When a copy of the descriptor is re-propagated by p_j, at time t_g, its age gets incremented by $\delta = t_g - t_a$. Finally, a global parameter of the algorithm named *propagation age limit* (*PAL*) defines an age threshold after which descriptors get discarded from the views of the nodes.

The age of a descriptor does not take into account the network delay time. This fact might affect the system performance if the network delay is not negligible with respect to δ, which in turn depends on the gossip period. Intuitively, a small *PAL* ensures therefore quick reaction to churn. On the other hand, it slows down convergence by reducing the lifetime of descriptors, regardless if they refer to faulty or alive processes.

This behavior can be partially mitigated through a sensible choice among equivalent descriptors during the merging phase. Equivalent descriptors refer to the same node and have the same logical clock, but since the age is monotonically increasing, a descriptor with higher age contains a more up-to-date approximation of the real utility value of the node. Naturally, this consideration holds under the assumption of a reasonable drift between clocks of different nodes.

Changes in the utility value of nodes in the eligible set $\mathcal{E}$ can be regarded as a mitigated form of churn: when a node becomes eligible, it starts emitting descriptors, while a node leaving $\mathcal{E}$ stops doing that. From the RANKSLICING protocol perspective, these events correspond to nodes appearing and disappearing respectively, although the underlying overlay is not affected.

If a node p_i leaves the $\mathcal{E}$ set, all the previously emitted descriptors (if any) are no longer replaced by fresh ones. If p_i was a supernode, the descriptors stored in the views will be eventually removed, as their age exceed the *PAL* threshold. Descriptors from the top eligible nodes will take their place during the following merge operations. This behavior matches the **Adaptiveness** requirement.

C. Algorithm improvements

Once received the first transmission from p_i, node p_j partially knows the content of the view of p_i. The answer sample can give priority to the locally stored descriptors known to be more up to date with respect to the corresponding ones owned by p_i. All the remaining empty slots of the answer sample are filled by taking random elements from the view, yet avoiding the items which are known to be owned by the initiator. The gossip session is terminated with the process p_i merging the answer as p_j did before. This is more effective than doubling the gossip frequency, as it takes advantage of the context provided by the inbound sample.

D. Algorithmic definition

We will refer to p_i as the local node (executing the algorithm). The internal state of p_i is defined by two dictionaries: *view* represents the view of the node p_i mapping nodes to their descriptors, while *tstamps* maps each key of *view* into the timestamp registered when the view entry was created.

For this purpose we define a set of methods for accessing a dictionary: Put(), Get(), Del(), KeysOf() and ValuesOf(). With the function EmitDescriptor() we represent the action of creating a fresh descriptor, characterized by an incremental logical clock value and age equal to 0. IsEligible() is a shorthand for $\mathsf{ep}(C^t(p_i))$, while the Now() function yields the current time-stamp. Methods Age() and UpdateAge() read and write the age field of a descriptor, respectively, while method ID() reads its identifier.

Each gossip session is characterized by the following steps:

1) The initiator queries the underlying topology management system for a random sample from the local neighborhood;
2) A sample of the view of size *size* is prepared by calling function PrepareSample($\emptyset$, *size*) (Algorithm 1) The output of the function is sent to the selected neighbor.
3) The neighbor receives a sample which is first passed as parameter to the Merge() function (Algorithm 2), then it generates an answer by passing the received sample S as the actual parameter for the PrepareSample(S, *size*) function.
4) Finally the initiator receives the answer sample, and runs the Merge() function on it.

Algorithm 1 shows the PrepareSample() function. The first part of the algorithm (lines 1 to 12) manages the emission of a local descriptor: eligibility gets verified and a descriptor is possibly inserted into the outbound sample. The view is maintained consistent accordingly, either updating or removing the local identifier stored in it, if any. The central part (lines 13 to 26) is executed only if a non-empty sample from the initiator is passed as parameter: prioritized elements of the local view, according to the logic described in Subsection IV-A are added to the outbound sample. Finally, the last part of the algorithm (lines 27 to 38) fills the remaining available space in the outbound sample with randomly taken elements. Only descriptors that have not expired are inserted into the sample.

Algorithm 2 shows the Merge() function. The *mergeMap* dictionary is filled with a fresh local descriptor (lines 4-5) and with all elements of the view (lines 6-11). All the elements from the inbound sample to be merged are added through the PutFresh() procedure (line 14 and Algorithm 3), which ensures that the most up-to-date descriptor is maintained for each referenced node. The descriptors are sorted with the SortByRanking() function (line 16), which implements a sorting of a set of descriptors based on the defined total ordering. Finally the state of the local node is rebuilt (lines 17-24).

Algorithm 1: PrepareSample(*neighborSample*, *size*)

```
1  ids ← view.Keys();
2  now ← Now();
3  if IsEligible(p_i) then
4      D ← EmitDescriptor();
5      sample ← {D};
6      if i ∈ ids then
7          view.Put(i, D);
8  else
9      sample ← ∅;
10     if i ∈ ids then
11         view.Remove(i);
12     ids ← ids \ {i};
13 if neighborSample ≠ NIL then
14     foreach D ∈ neighborSample do
15         j ← D.getID();
16         if j ∈ ids then
17             D' ← view.Get(j);
18             age ← D'.Age() + now − tstamps.Get(j);
19             if D'.lc() > D.lc() and age ≤ PAL then
20                 D'.UpdateAge(age);
21                 tstamps.Put(j, now);
22                 sample ← sample ∪ {D'};
23             else
24                 tstamps.Del(j);
25                 view.Del(j);
26             ids ← ids \ {j};
27 while |sample| < size & ids ≠ ∅ do
28     j ← RandomSample(ids);
29     ids ← ids \ {j};
30     D ← view.Get(j);
31     age ← AgeOf(D) + now − tstamps.Get(j);
32     if age ≤ PAL then
33         D.UpdateAge(age);
34         tstamps.Put(j, now);
35         sample ← sample ∪ {D_j^(l)};
36     else
37         tstamps.Del(j);
38         view.Del(j);
39 return sample;
```

Algorithm 2: Merge(*neigborSample*)

```
1  mergeMap ← new Map();
2  now ← Now();
3  if IsEligible(p_i) then
4      D ← EmitDescriptor();
5      Put(mergeMap, D);
6  foreach D ∈ view.Values() s.t. D.getID(≠)i do
7      j ← D.getID();
8      age ← D.AgeOf() + now − tstamps.Get(j);
9      if age ≤ PAL then
10         D.UpdateAge(age);
11         mergeMap.Put(D);
12 foreach D ∈ neighborSample do
13     if j ≠ i then
14         PutFresh(mergeMap, D);
15 candidates ← mergeMap.Values();
16 sorted ← candidates.SortByRanking();
17 nextView ← new Map();
18 nextTstamps ← new Map();
19 for idx ∈ [1 ··· Min(K, |sorted|)] do
20     D ← sorted[idx];
21     nextView.Put(j, D);
22     nextTstamps.Put(j, now);
23 view ← nextView;
24 tstamps ← nextTstamps;
```

Algorithm 3: PutFresh(*mergeMap*, D)

```
1  if j ∈ mergeMap.Keys() then
2      D' ← Get(mergeMap, j);
3      if D.lc() > D'.lc() then
4          mergeMap.Put(j, D);
5  else
6      mergeMap.Put(j, D);
```

E. Pragmatic Aspects

During the experimentation phase, a behavioral difference between nodes in the open Internet and NATed nodes emerged from the dataset. We noticed that the former group show a much quicker convergence with respect to the latter one. The effect of this phenomenon is captured in Figure 7.

This phenomenon is associated to how WPSS [18], that is the peer sampling system we used for both the simulations and the real deployment, constructs the NAT-resilient overlay network. Specifically, in WPSS public nodes have higher number of neighbors with respect to NATed ones. As consequence, a NATed node has fewer possible candidates for gossiping.

This is not a structural problem, as private nodes are still able to converge to the correct supernode set. However, we provide a modification of the algorithm to let private peers converge as quick as the open Internet ones, we call that *overriding procedure*.

This procedure works in the following manner: once the perceived quality measure of an open Internet node reaches a certain threshold value, reasonably close to 1, an *override request* containing its view and perceived quality is sent to the subset of neighbors behind a NAT. A node receiving an override request executes a merge operation as for a normal gossip session, although no answer is generated. This strategy introduces a new parameter, called *OQT* (Overriding Quality Threshold).

F. Supernode set estimation quality

The view maintained by each node corresponds to an approximation of the supernode set that the application can access locally both on the supernodes and on the other peers.

We define a quality measure q of the approximated supernode set with values in the $[0, 1]$ interval. A value of q close to 1 means that the view of a node is equal to the optimal supernode set of size K.

We define then the *actual quality* of the system, which represents a general measure of the quality of the supernode set estimation in the whole system. The actual quality is an average computed by retrieving the view V_i of each node p_i,

and comparing it with an ideal supernode set L in the following way:

$$q = \frac{1}{N} \sum_{p_i \in \Pi} \frac{|V_i \cap L|}{k}$$

Such computation can be easily achieved when having global knowledge of the system, such as in simulation, by accessing the view of the nodes and comparing it with the optimal set of supernodes L. This is however not feasible in a real deployment. For that reason, we emulate global knowledge by letting peers report their current state to a central server and then we compute the quality there.

We define also another measure of quality of the supernode set that is estimated in a distributed manner: the *perceived quality*. This measure of quality is made available to the application using our protocol, to determine how good is the approximation of the supernode set in order to avoid executing expensive operations whenever the quality is low.

The estimation of the perceived quality is based on the assumption that the local view of each node increases in accuracy at every gossip session and it is executed each time a node p_i obtains a new view V_i' from merging an incoming sample with its current view V_i. The first step of the calculation by comparing the current view with the view obtained by merging the received sample:

$$q_{i,0} = \frac{|V_i \cap V_i'|}{K}$$

The resulting quality value $q_{i,0}$ is subject to quick fluctuations, as it changes at every gossip round according to how similar the local view is to the view of the neighbor. A second quality evaluation $q_{i,1}$ is therefore estimated through an iterative moving average:

$$\begin{aligned} q_{i,1}^{(0)} &= 0 \\ q_{i,1}^{(n)} &= \alpha \cdot q_{i,1}^{(n-1)} + (1-\alpha) \cdot q_{i,0} \qquad \alpha \in [0,1] \end{aligned}$$

The base of the iterative computation $q_{i,1}^{(0)}$ is zero because initially each view is empty, hence the intersection with the ideal supernodes set would be empty. Intuitively, the perceived quality is an optimistic measure, as the quality values are based on pieces of information which come from the direct neighborhood.

V. Experimental results

In this section, we delve into the experimental evaluation of RankSlicing, at first in a simulated environment, then on an actual deployment.

A. Experimental setup

We implemented our protocol in a production-quality development framework used also for our commercial products. That is an event-driven and component-based framework which enables code to be executed both in simulation and in a real deployment. The tool allows to emulate a number of network characteristics in simulation such as delays, connectivity limitations given by the presence of NATs, and bandwidth allocation dynamics. Every experiment in simulation was run at least 20 times for statistical significance, and in four different churn scenarios:

- **C00** No churn;
- **C03** Churn 0.003 (meaning that 0.3% of the nodes leave the system and get replaced within 10s);
- **C05** Churn 0.005 (0.5% every 10s);
- **C10** Churn 0.010 (1.0% every 10s);

In real deployment, we deploy our protocol on around 6500 of consumer machines. Those are provided by users of our commercial live streaming application who allowed us to run experiments on their computers. These hosts are located mostly in Sweden (80%) but also in Europe (12%) and in the US (7%). The ratio of open-Internet nodes is 20% while the rest of the hosts are behind NAT.

All experiments were conducted using an agent installed in our volunteers' machine which can be instrumented remotely to run parameters study. Peer-to-peer communication in this setting was provided by the framework using UDP-based network library which features the same reliability, congestion and flow control as TCP, plus it offers NAT traversal capabilities.

As NAT-aware topology management of choice, we use WPSS both in simulation and in the real world scenario [18].

B. Methodology

We first study the system behavior in simulation, where we strive to identify the best set of values for the aforementioned parameters by recreating similar conditions to the ones of our real deployment. For that, we configure the simulator to emulate the same estimated delay, bandwidth and connectivity success probability observed in our consumer test network.

After that, in deployment, we configure our protocol with the best set of values obtained in simulation, always for the parameters described above, and assert the correct functioning of the protocol.

In all our experiments, both in simulation and deployment, we used an overlay size of $N = 1,000$ peers and a gossip session period T of 1 second. Regarding the ranking of peers, we chose to let the utility function assign each node a random value in the $[0,1]$ range. That value is kept constant for the whole execution of the test. We do not explore variations of the utility value of a peer over time but instead we concentrate on churn given that it produces the same kind of phenomena on our algorithm.

In simulation, we use both perceived and actual quality as main performance metrics for our protocol, as we defined in Subsection IV-F. In order to compute the actual quality, we compare the supernode set of each node with a "perfect" supernode set built from the global view of the system. After that, we average the actual quality values of all nodes for obtaining a single measure of actual quality.

In deployment, we also provide both perceived and actual quality. The former is reported by the peers to a snapshot server over time. The second instead is calculated by having peers send the totality of their supernode set to the snapshot server

and then comparing that set with the ideal set of nodes derived from a central ranking of nodes constructed at the snapshot server.

C. Evaluation in simulation

In this section, we evaluate the performance of our protocol in simulation.

1) Sampling size: We start by studying the behavior of our protocol with different values of the sampling size, identified by the parameter H. In the same set of experiments, we assert whether the behavior of the protocol remains consistent for different values of K. The H parameter defines the maximum number of elements transmitted for each sample. Values valid for H are $[1..K]$. In general, we expect that for values of H that are closer to K to obtain faster convergence since most or all of the supernode set will be transferred between peers. As we will see, that will come at a higher network bandwidth usage.

In order to assess the performance of the protocol, we observe the evolution of the absolute quality in 20 different experiments from the following parameter space:

- $K \in \{ 50, 10 \}$
- $H/K \in \{ 0.1, 0.2, 0.3, 0.4, 0.5, 1 \}$
- $PAL = 9500$ms (chosen arbitrarily)

Besides the actual quality, we monitor the evolution of the perceived quality in presence of churn and for all churn classes mentioned earlier. Table I reports the analysis of the time required to converge to the 90% of the steady state quality. This yields a first estimation of the convergence time. We choose 90% because, according to our experience in P2P live streaming, it is a reasonable performance level for our system to behave correctly.

We can observe that the best performance on both actual and perceived quality is achieved when $H/K = 1$, namely when the initiator of a gossip session shares all its view. From the analysis we can also see that we are still able to reach convergence in a reasonable amount of time for H/K in the $[0.3, 1)$ range. However, convergence time significantly increases as values of H/K get closer to zero. This can be seen graphically in Figure 1, which shows the reported result for the **C03** churn class.

In Table II, we report an analysis of the bandwidth utilized by our algorithm, in bytes per second, in the same set of experiments described above. Considering that the H parameter defines the size of the sample, we obtain a linear decrease of bandwidth utilization when the H/K ratio also decreases. From the table we can also see that bandwidth utilization is also partially influenced by both the churn and the parameter K.

2) Refinement of the perceived quality: As mentioned in Section IV-F, the perceived quality metric yields a less accurate estimation than the actual quality metric.

The α parameter represents the smoothing factor for the perceived quality computation. The valid range for it is in the $[0, 1)$ interval: as the value of α is set closer to 1 we obtain a higher hysteresis. By increasing α, we obtain that nodes perceive a slower convergence to the optimal value 1, which

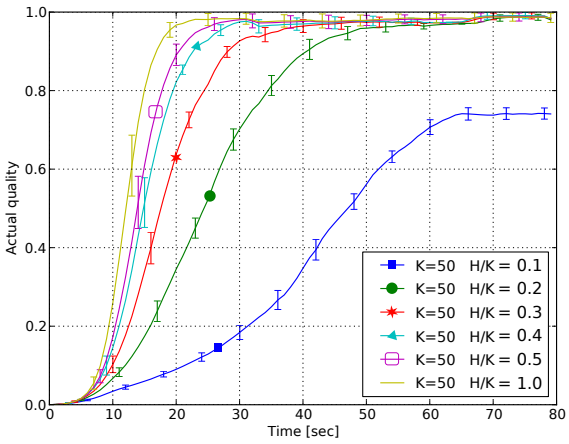

Fig. 1. Evolution of quality for different ratios of H over K and $K = 50$. The curves show the average actual quality, while the whiskers show the standard deviation. The graph refers to experiments belonging to the churn class **C03** (0.3% nodes leaving and joining the system within 10 seconds) The other churn classes show similar behaviors, summarized in Table I

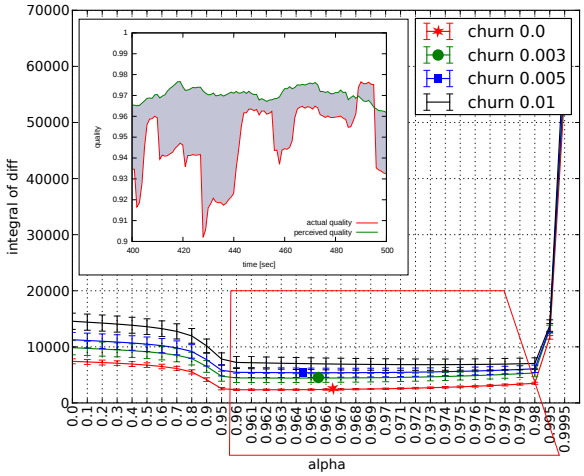

Fig. 2. Identification of the best value for the α parameter: the process consists in computing, for each experiment, the integral of the absolute difference between perceived and actual quality (corresponding to the grey area on the inner graph). By minimizing the value of the integral we identify the configuration of α which maximizes the precision of the perceived quality.

corresponds to the actual quality of the system. All peers start in a situation where peers have perceived quality $q = 0$ then the quality increases until it converges to a stable value, this in case of absence of churn. However, in the presence of churn, high values of α reduces the capacity of the protocol to detect the changes in the supernode set introduced by the churn.

In this set of experiments, shown in Figure 2, we strive to find the value of the α parameter which yields the best estimation of the actual quality.

For each experiment, we compute the difference between the actual quality and perceived quality measures in time, and integrate the results to observe the speed of the changes: smaller integral corresponds to a smaller difference area between the two measures, hence a minimization of the integral

	K	10				50			
Churn	h/K	1	0.5	0.4	0.3	1	0.5	0.4	0.3
C00 (0.0 %)	μ	18.0792	22.3065	26.0538	32.0598	17.8289	20.7665	23.0508	26.5543
	σ	1.5378	1.8242	1.4640	1.9527	0.8055	0.9344	0.6879	0.4719
C03 (0.3 %)	μ	17.8289	22.6915	26.0538	33.7282	17.9791	20.7280	22.3835	27.2217
	σ	1.2043	1.5953	1.0683	1.8653	0.8315	0.9078	0.5005	0.7461
C05 (0.5 %)	μ	18.2294	22.4220	26.7212	33.3945	17.8790	20.5740	23.0508	27.2217
	σ	1.1090	1.4751	2.2692	2.2196	0.7078	0.7223	0.6879	0.9438
C10 (1.0 %)	μ	19.3805	24.5010	29.5573	36.2307	18.0291	20.7280	23.3845	28.7232
	σ	5.8153	5.5808	5.1906	6.1886	0.7929	0.8643	2.1430	1.9527

TABLE I. DIFFERENT CHOICES OF K AND H/K INFLUENCE THE CONVERGENCE TIME OF THE SYSTEM. THE REPORTED MEASURE REFERS TO THE AVERAGE TIME REQUIRED TO REACH THE 90% OF THE STEADY STATE QUALITY.

		$K = 50$				$K = 10$			
Churn	h/k	Inbound μ (bytes/sec)	Inbound (ratio)	Outbound μ (bytes/sec)	Outbound (ratio)	Inbound μ (bytes/sec)	Inbound (ratio)	Outbound μ (bytes/sec)	Outbound (ratio)
C00 (0.0%)	1	227.5421	1.0000	100.4379	1.0000	54.2266	1.0000	24.0302	1.0000
	0.5	120.5770	0.5299	52.7411	0.5251	31.4794	0.5805	14.4576	0.6016
	0.4	98.8093	0.4342	43.1918	0.4300	26.8249	0.4947	12.5427	0.5220
	0.3	76.7182	0.3372	33.6207	0.3347	22.1346	0.4082	10.6246	0.4421
C03 (0.3%)	1	226.2513	1.0000	96.7191	1.0000	53.6728	1.0000	23.2650	1.0000
	0.5	119.7833	0.5294	50.8883	0.5261	31.0439	0.5784	14.0683	0.6047
	0.4	98.1803	0.4339	41.7576	0.4317	26.4386	0.4926	12.2446	0.5263
	0.3	76.1631	0.3366	32.5417	0.3365	21.7685	0.4056	10.3974	0.4469
C05 (0.5%)	1	225.4849	1.0000	94.3091	1.0000	53.3811	1.0000	22.7784	1.0000
	0.5	119.2337	0.5288	49.6770	0.5267	30.7802	0.5766	13.8182	0.6066
	0.4	97.5229	0.4325	40.7761	0.4324	26.1177	0.4893	12.0296	0.5281
	0.3	75.5962	0.3353	31.7993	0.3372	21.4730	0.4023	10.2305	0.4491
C10 (1.0%)	1	223.5101	1.0000	87.8138	1.0000	52.5535	1.0000	21.4467	1.0000
	0.5	117.8882	0.5274	46.3856	0.5282	30.0637	0.5721	13.1227	0.6119
	0.4	96.0894	0.4299	38.0657	0.4335	25.3695	0.4827	11.4365	0.5332
	0.3	74.3708	0.3327	29.7503	0.3388	20.7698	0.3952	9.7697	0.4555

TABLE II. AVERAGE BANDWIDTH USAGE IN 4-MINUTES LONG EXPERIMENTS. THE *Inbound ratio* AND *Outbound ratio* COLUMNS SHOW THE BANDWIDTH USAGE NORMALIZED OVER $H/K = 1$.

gives the best value for the α parameter.

We set the values of $K = 50$, $H/K = 0.5$ which are the best values identified in the previous experiments and $PAL = 9500ms$, which will see is also the best value for that metric. We ran 20 tests for $\alpha \in [0, 1)$, analyzing the behavior in the four churn classes. After a preliminary study we realized that the perceived quality measure works better for values in the $[0.95, 0.98]$ range, while it abruptly worsens as values get closer to 1. This fact is depicted in Figure 2, which focuses on the behavior in the best range: we can see that the system follows similar dynamics for all the churn classes we took in account.

3) Identification of Propagation Age Limit (PAL): The Propagation Age Limit defines the maximum age threshold for descriptors. Intuitively, the value of *PAL* should be greater than T, otherwise node descriptors would not be propagated at all. Small values translate into a faster elimination of descriptors of potentially failed nodes, but also in a slower convergence time, since good descriptors are also quickly removed from the system. Here we show the results obtained from the analysis of this trade-off.

We ran two sets of experiments, the first consisted in running the algorithm with the various churn classes, while in the second set of experiments the scenario scheduled a catastrophic churn effect, with the 20% of the nodes leaving the network simultaneously, after the supernodes set is built.

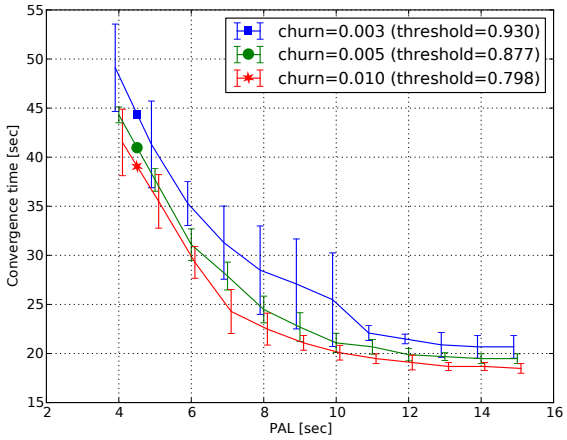

Fig. 3. Relation between the *PAL* parameter and the average time required to converge to a steady state quality. Higher values of *PAL* allow the descriptors of the super-nodes to survive and get propagated for more time.

In both settings, we started from a base configuration of $K = 50$, $H/K = 0.5$, $\alpha = 0.95$, varying the *PAL* parameter in the $[9000, 15000]$ (milliseconds) range.

In the first experiment, we identify how the *PAL* parameter influences the convergence time. The analysis involved two

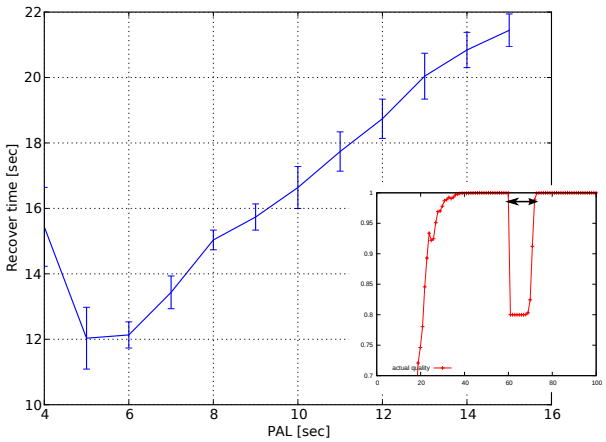

Fig. 4. Relation between the *PAL* parameter and the time to recover from a catastrophic churn event (shown in the subgraph). High values result in a slower recover from failure, as all descriptors, including the ones referring disappeared supernodes, remain in the system for a longer time.

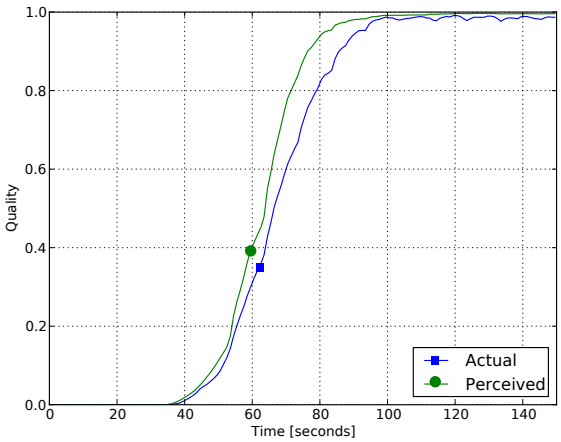

Fig. 5. Quality evolution in time of the deployed algorithm (single experiment). The curves show the average actual quality of the nodes running the algorithm.

steps: the isolation of a target quality threshold for each churn class (achieved by measuring the minimum quality value when the system has reached a stable quality) and the aggregation by churn class of the convergence times to the defined threshold (average and standard deviation). In Figure 3 we can see that, for this setting, a *PAL* value greater than 12 seconds results in better convergence performance.

The second set of experiments aims at measuring the effects of the *PAL* parameter on the recovery from churn effects. The analysis has been achieved in the same way as the first set of experiments, but measuring the time required to recover to a stable quality value after a catastrophic churn event. We can see in Figure 4 that the recovery time grows linearly with the *PAL*. This because the descriptors referring to faulty nodes get removed and replaced only when their age reach the value of *PAL*.

The picture also highlights a non-linear behavior for values of *PAL* smaller than 6 seconds. This is due to the fact that, for those values, descriptors referring to working supernode get removed from the views before they can be refreshed.

D. Evaluation in deployment

The experiments executed on the real deployment involved a 1 000 nodes sample out of around 6 500 available hosts. Note that the network was subject to natural churn, that is the number of nodes was decreasing in time during any given experiment because of users turning off their machines. Besides that, the experiment could not be started on around 8% of the hosts due to firewall limitations and congestion in the network.

For deployment experiments, we set $H = K$ and $K = 10$. As a consequence, the actual quality can only assume 10 values in the range [0,1]. Based on the simulation results, we select a value for α of 0.95 and a value for the *PAL* of $12s$. The overriding quality threshold, *OQT*, was set to 0.975. We let all peers join uniformly at random in the first minute of the test, the algorithm is started on the node 30 seconds after the join

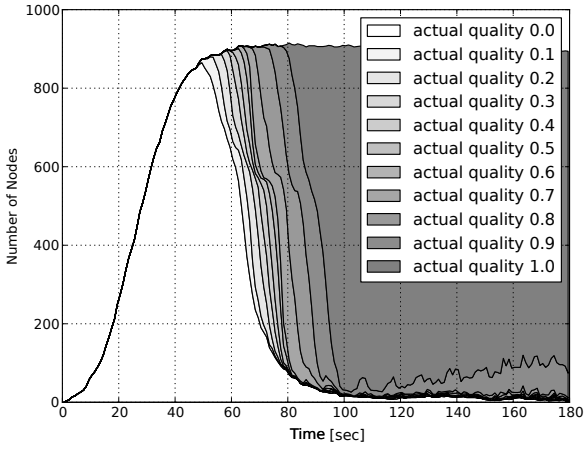

Fig. 6. Average distribution of actual quality for the deployed algorithm (8 experiments). Each area represents the absolute number of nodes having the same quality value.

such to allow time for the underlying topology, provided by WPSS, to be set up.

Figure 5 shows the evolution of the average perceived quality in deployment. The same experiment is reported in Figure 6 from the point of view of quality distribution among nodes over time. In the Figure, the upper line represents the total number of peers. The colored areas beneath that line indicate how many of those peers experience a specific value of actual capacity. As we can see, almost all nodes achieve an actual quality value close or above 90% after 20 seconds that all peers have joined.

We record also the time required to reach the steady state of the perceived quality values of 90% and 97.5%. The measure was taken for each node, relative to the time that it first reported to the snapshot server. Table III shows the related statistics.

In Figure 7, we show the difference between the actual and

Quality Threshold Percent	Quality Threshold Value	Convergence Time (seconds) μ	σ
90%	0.874442513738	56.4706454741	11.4602132189
97.5%	0.947312723216	62.2236587216	12.1045159194

TABLE III. CONVERGENCE TIME OF A SET OF 1 000 NODES AS EXPERIMENTED IN A REAL DEPLOYMENT. THE REPORTED VALUE IS RELATIVE TO SINGLE NODES AND REFERS TO THE TIME REQUIRED TO REACH 90% AND 97.5% OF THE STEADY-STATE ACTUAL QUALITY.

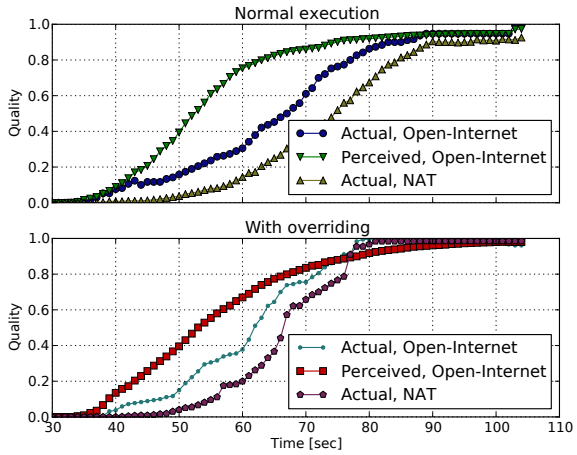

Fig. 7. Comparison of convergence time for open Internet and NATed nodes, in actual deployment. The graph on the top highlights the difference in performance for one normal execution of 100 nodes. The graph on the bottom shows a second execution, in which the overriding procedure is triggered on the open Internet nodes, with an Overriding Quality Threshold of 0.97.

perceived quality measurements at NATed and open-Internet nodes. As we can see, thanks to the variant of the algorithm described in Section IV-E, both set of nodes reach the same quality values over time.

VI. CONCLUSIONS

In this paper, we proposed RANKSLICING, a decentralized algorithm for supernode selection that allows to identify the best K nodes in the overlay. Our approach consists of an epidemic protocol that is highly resilient to churn and does not require new connections to be established, but rather relies on established connections, such as the ones provided by a NAT-resilient peer sampling framework. Thorough experimental analysis both in simulation and on a deployment of thousands of consumer machines shows that the solution is practical and meets the requirements of consistency and stability imposed by our use-case, that is a P2P live streaming application.

As future work, we will be incorporating RANKSLICING into our application and study its behavior together with the rest of the system.

VII. ACKNOWLEDGEMENTS

This work was partially supported by the European Union, specifically by the FP7 EU project iSocial (FP7-ITN-316808).

REFERENCES

[1] S. A. Baset and H. Schulzrinne. An analysis of the skype peer-to-peer internet telephony protocol. *arXiv preprint cs/0412017*, 2004.

[2] J. Dowling and A. H. Payberah. Shuffling with a croupier: Nat-aware peer-sampling. In *Proc. of the 32nd Int. Conf. on Distributed Computing Systems (ICDCS'12)*, pages 102–111. IEEE, 2012.

[3] P. Felber and E. W. Biersack. Cooperative content distribution: Scalability through self-organization. In *Self-star Properties in Complex Information Systems*, pages 343–357. Springer, 2005.

[4] A. Fernández, V. Gramoli, E. Jiménez, A.-M. Kermarrec, and M. Rayna. Distributed slicing in dynamic systems. In *Proc. of the 27th Int. Conf. on Distributed Computing Systems (ICDCS'07)*, page 66. IEEE, 2007.

[5] C. Huang, J. Li, and K. W. Ross. Peer-assisted VoD: Making internet video distribution cheap. In *Proc. of the 6th Int. Workshop on Peer-to-Peer Systems (IPTPS'07)*, 2007.

[6] M. Jelasity and A.-M. Kermarrec. Ordered slicing of very large-scale overlay networks. In *Proc. of the 6th IEEE Int. Conf. on Peer-to-Peer Computing (P2P'06)*, pages 117–124. IEEE, 2006.

[7] M. Jelasity, A. Montresor, and O. Babaoglu. Gossip-based aggregation in large dynamic networks. *ACM Transactions on Computer Systems (TOCS)*, 23(3):219–252, 2005.

[8] A.-M. Kermarrec, A. Pace, V. Quema, and V. Schiavoni. NAT-resilient gossip peer sampling. In *Proc. of the 29th Int. Conf. on Distributed Computing Systems (ICDCS'09)*, pages 360–367. IEEE, 2009.

[9] M. Liu, E. Harjula, and M. Ylianttila. An efficient selection algorithm for building a super-peer overlay. *Journal of Internet Services and Applications*, 4(1):1–12, 2013.

[10] A. Mansy and M. Ammar. Analysis of adaptive streaming for hybrid cdn/p2p live video systems. In *Proc. of the 19th Int. Conf. Network Protocols (ICNP'11)*, pages 276–285. IEEE, 2011.

[11] A. Montresor and R. Zandonati. Absolute slicing in peer-to-peer systems. In *Proc. of the Int. Symposium on Parallel and Distributed Processing (IPDPS'08)*, pages 1–8. IEEE, 2008.

[12] S. Niazi and J. Dowling. Usurp: Distributed NAT traversal for overlay networks. In P. Felber and R. Rouvoy, editors, *Distributed Applications and Interoperable Systems*, volume 6723 of *LNCS*, pages 29–42. Springer Berlin Heidelberg, 2011.

[13] A. H. Payberah, J. Dowling, F. Rahimian, and S. Haridi. gradienTv: Market-based p2p live media streaming on the gradient overlay. In *Distributed Applications and Interoperable Systems*, volume 6115 of *LNCS*, pages 212–225. Springer, 2010.

[14] K. Ponnavaikko and D. Janakiram. The edge node file system: A distributed file system for high performance computing. *Scalable Computing: Practice and Experience*, 10(1), 2001.

[15] Y. Qiao and F. E. Bustamante. Structured and unstructured overlays under the microscope. *USENIX Annual Technical Confrence*, 2006.

[16] V. Raychoudhury, J. Cao, and W. Wu. Top k-leader election in wireless ad hoc networks. In *In Proc. of 17th Int. Conf. on Computer Communications and Networks (ICCCN'08)*, pages 1–6. IEEE, 2008.

[17] M. Ripeanu. Peer-to-peer architecture case study: Gnutella network. In *Proc. of the 1st Int. Conf. on Peer-to-Peer Computing (P2P'01)*, pages 99–100. IEEE, 2001.

[18] R. Roverso, J. Dowling, and M. Jelasity. Through the wormhole: Low cost, fresh peer sampling for the internet. In *Proc. of the 13th Int. Conf. on Peer-to-Peer Computing (P2P'13)*, 2013.

[19] R. Roverso, S. El-Ansary, and M. Högqvist. On http live streaming in large enterprises. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 489–490, New York, NY, USA, 2013. ACM.

[20] J. Sacha, J. Dowling, R. Cunningham, and R. Meier. Discovery of stable peers in a self-organising peer-to-peer gradient topology. In *Distributed Applications and Interoperable Systems (DAIS'06)*, volume 7891 of *LNCS*, pages 70–83. Springer, 2006.

[21] O. D. Sahin, C. E. Gerede, D. Agrawal, A. El Abbadi, O. Ibarra, and J. Su. Spider: P2p-based web service discovery. In *Service-Oriented Computing (ICSOC'05)*, volume 3826 of *LNCS*, pages 157–169. Springer, 2005.

[22] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Comput. Commun. Rev.*, 31(4):149–160, 2001.