Absolute Slicing in Peer-to-peer Systems

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

Roberto Zandonati
University of Trento, Italy
r.zandonati@studenti.unitn.it

Abstract

Peer-to-peer (P2P) systems are slowly moving from application-specific architectures to a generic service-oriented design framework. The idea is to allow a dynamic collection of P2P applications to cohabit into a single system, with applications starting and terminating at will, or even changing their requirements at run-time. This raises an interesting problem in connection with managing resource assignment in a large-scale, heterogeneous and unreliable environment. Recently, the distributed slicing service *has been proposed to allow for an automatic partitioning of P2P networks into groups (*slices*) that represent a controllable amount of some resource. A particular instantiation of such service has been described, called* ordered slicing*, in which nodes are ranked based on some metrics and then assigned to a slice based on their position in the ranking. In this paper, we present an alternative version of the problem called* absolute slicing*. Here, the goal is to assign a specified number of nodes to a slice and maintain such assignment in spite of churn. We propose a simple algorithm that solves the problem by combining well-known protocols such as peer sampling and aggregation, and we experimentally evaluate its performance.*

1 Introduction

Peer-to-peer (P2P) protocols have proven efficient to provide scalable solutions for the implementation of large-scale distributed applications, successfully coping with unreliability and dynamism. Yet, the current state of affairs has failed to relax one important restriction: P2P protocols are often very specific to a given application. As a result, a file-sharing system is only adapted for file sharing, a desktop grid is able to execute only specially tailored and centrally managed tasks, and so on. This situation is somewhat comparable to having a powerful computer that can run only one application, without the possibility of reprogramming to exploit all its potentials.

To become a mature technology, the P2P community must take one step further: move away from application-specific architectures towards a generic, service-oriented design paradigm. The idea is to integrate the P2P approach into distributed platforms on top of which several applications may cohabit. Applications may change their requirements at runtime; new ones may be added and existing ones may be removed. Furthermore, we should be able to merge systems on which P2P applications are running, or split existing ones. Such flexibility would allow us to deploy P2P applications much easier and let several such applications to co-exist independently while making use efficiently of shared underlying resources and protocols. Examples of such platforms include testbeds such as PlanetLab [13], desktop-grid-like applications [11, 14], and potentially even networks of set-top-boxes owned by a single ISP [2], but physically located at clients' houses.

A possible framework to implement such ideas has been described in [2], together with a list of open research problems. Among the issues to be solved, an important one is how to allocate a set of nodes for a given application.

The general problem has been recently named *distributed slicing* [5, 7] and may be described as follows: given a distributed collection of nodes characterized by one or more attributes, we want to allocate a subset ("slice") of them to a specific application, by selecting those that satisfy a given condition over the attributes and maintaining such allocation in spite of churn.

A specific instance of this problem, called *ordered slicing*, has been recently introduced and solved [5]. In ordered slicing, nodes are ranked according to their capabilities expressed by an attribute value, and a given percentage of them is assigned to a specific application, based on this ranking. For example, we could assign the best connected nodes to a superpeer slice by selecting the top 10% nodes ordered by their bandwidth.

In this paper, we propose another variant of the distributed slicing problem, called *absolute slicing*. Here, applications may require to be assigned a given *quantity* of nodes from a larger collection whose elements satisfy a particular condition over attributes.

As an example of context and motivation for such prob-

lem, consider a desktop grid application where a customer want to rent a specific amount of distributed storage. The actual request could be: give me 1000 machines with at least 1GB of available storage and connected through a T1 line or higher.

Due to the intrinsic dynamism of contemporary P2P systems, it is impossible to obtain accurate information about the capabilities (or even the identity) of the system participants. Consequently, no node is able to maintain accurate information about the entire system. This disqualifies centralized approaches. For this reason, we propose here a decentralized solution to the absolute slicing problem, based on the gossip paradigm [4]. In recent years, the label "gossip" has been applied to an increasingly larger class of algorithms, going outside the original and limited field of information dissemination [4]. Gossip-based approaches exist now for information aggregation, overlay network management and clock synchronization, just to cite a few [1, 6, 8]. Their distinctive features include relying only on local information and being extremely robust.

Our solution is based on two well-known components: *peer sampling* [10] and *decentralized aggregation* [8]. Peer sampling provides nodes with continuously up-to-date random samples of a specific population of nodes; higher-level gossip protocols use these samples to contact random peers. Decentralized aggregation computes an aggregate function, such as average, sum or counting, over the population of nodes associated with the peer sampling layer. The idea is to identify the group of nodes that satisfy a given condition (through peer sampling), and then probabilistically select nodes until the required quantity is reached (measured through aggregation).

The rest of the paper is organized as follows. The system model is described in Section 2. Section 3 provides background information and discusses related work. The specification of the problem is introduced in Section 4. The architecture of our solution, including how existing techniques are used and/or adapted, is described in Section 5. Experimental results are shown in Section 6. Finally, Section 7 draws the conclusions and discusses potential future work.

2 System Model

We consider a collection of nodes that are connected through an underlying routed network, such as the Internet, where every node can potentially communicate with every other node. To actually communicate, a node has to know the address of another node. This is achieved by maintaining a (*partial*) *view* at each node that contains a set of node descriptors. Views can be interpreted as sets of edges between nodes, naturally defining a directed graph over the nodes that determines the topology of an *overlay network*.

// Active Thread:
during every cycle do
$q \leftarrow getPeer()$
send $(prepareMessage(s_p), request, p)$ to q

// Passive thread:
do forever
$(m_q, type, q) \leftarrow$ receive(*)
if $type = request$ **then**
send $(prepareMessage(s_p, m_q), reply, p)$ to q
$s_p \leftarrow update(s_p, m_q)$

Figure 1. Generic gossip protocol executed by node p. The local state of p is denoted as s_p. The message sent by q is denoted as m_q.

The network is highly dynamic; new nodes may join at any time, and existing nodes may leave, either voluntarily or by *crashing*. Our approach does not require any mechanism specific to leaves: spontaneous crashes and voluntary leaves are treated uniformly. Thus, in the following, we limit our discussion to node crashes. Byzantine failures, with nodes behaving arbitrarily, are excluded from the present discussion.

Communication incurs unpredictable delays and is subject to failures. Single messages may be lost, links between pairs of nodes may break. Nodes have access to local clocks that can measure the passage of real time with reasonable accuracy, that is, with small short-term drift.

3 Background and Related Work

3.1 Gossip protocols

All the protocols implemented in this paper are based on the gossip paradigm [4] and follow the generic algorithmic skeleton illustrated in Figure 1. Our gossip protocols are composed by an *active thread*, that periodically sends a message to a peer node selected through the $getPeer()$ function, and a passive thread, that receives such messages and sends replies. We use the term *cycle* to denote the fixed time interval that occurs between two consecutive messages sent by the active thread. Messages contain a representation of the locale state of the sender, obtained through a generic $prepareMessage()$ function; their content is used to update the local state, which is done through the $update()$ function. These three functions are left unspecified here and will be later customized based on the particular functionality to be implemented.

3.2 Peer Sampling

A *peer sampling service* provides each node with continuously up-to-date random samples of a population of nodes. Such samples fulfill two purposes: locally, they provide each node with random peers that can be used to implement higher-level gossip protocols; globally, they define an *overlay topology*, i.e. a directed graph superimposed over the network and representing the "social" relationships among nodes. In the peer sampling case, the graph is characterized by a *random* structure and the presence of a single strongly connected component. There are already a large number of examples of exploitation of such service for the implementation of higher-level services and applications [6, 12, 15, 16].

In this paper we consider an instantiation of the peer sampling service based on the NEWSCAST protocol [10], chosen for its low cost, extreme robustness and minimal assumptions. The basic idea of NEWSCAST is that each node maintains a local set of random descriptors, called the (partial) *view*. A descriptor is a pair (*node address*, *timestamp*). Function $getPeer()$ returns a random member of the view; function $prepareMessage()$ returns the view itself, plus a fresh descriptor of itself. When receiving a message, function $update()$ keeps a fixed number of freshest descriptors (based on timestamps), selected from those locally available in the view and those contained in the message. Nodes belonging to the network continuously inject their identifiers in the network with the current timestamp, so old identifiers are gradually removed from the system and are replaced by newer information. This feature allows the protocol to "repair" the overlay topology by forgetting information about crashed neighbors, which by definition cannot inject their identifiers.

Implementations exist in which these messages are small UDP messages containing approximately 20-30 descriptors, each composed of an IP address, a port and a timestamp. The cycle length is typically long, in the range of 10 s. The cost is therefore small, few tens of bytes per second, similar to that of heartbeats in many distributed architectures. The protocol provides high quality (i.e., sufficiently random) samples not only during normal operation (with relatively low churn), but also during massive churn and even after catastrophic failures (up to 70% nodes may fail), quickly removing failed nodes from the local views of correct nodes.

3.3 Decentralized Aggregation

Aggregation is a common name for a set of functions that provide a summary of some global system property. In other words, they allow local access to global information in order to simplify the task of controlling, monitoring and optimizing distributed applications. Examples of aggregation functions include network size, total free storage, maximum load, average uptime, location and intensity of hotspots, etc.

In this paper, we adopt the gossip algorithm introduced in [8] to compute aggregates in a robust and decentralized manner. The algorithm assumes that each node maintains a local approximation of the aggregate value, initially equal to the value of the local property. Function $getPeer()$ returns a random node taken from the local view of an associated peer sampling. Function $prepareMessage()$ returns the current local approximate value, while function $update()$ modifies the local approximate value based on some aggregation-specific and strictly local computation based on the previous values. This local pairwise interaction is designed in such a way that all approximate values in the system will quickly converge to the desired aggregate value.

The algorithm proposed is *proactive* and *adaptive*; proactive means that all nodes participating in an aggregate computation are made aware of the final result, while adaptive means that aggregate information is kept continuously up-to-date in spite of dynamism of the network. In order to satisfy the latter requirement, the protocol needs to be periodically restarted; the final aggregate value is provided to higher-level protocols, while the approximate values are initialized again to the local property. The periodicity of restarting events is called *epoch*.

Among the algorithms described in [8], we will adopt COUNT, that computes the size of a random overlay network in logarithmic number of cycles w.r.t. to its size. To give an idea of the numbers involved, cycle length is around 10 s; messages contains around 20 numerical values; epochs normally corresponds to 20-30 cycles, a value sufficient for networks containing millions or even billions of nodes. As before, the overhead associated to this protocol is negligible.

3.4 Distributed Slicing

As far as we know, the distributed slicing problem was studied in a P2P system for the first time in [7]. In this paper, a node with the k^{th} smallest attribute value, among those in a system of size n, tries to estimate its normalized index k/n. The algorithm proposed in [7] works as follows. Initially, each node draws independently and uniformly a random value in the interval $(0, 1]$ which serves as its first estimate of its normalized index. Then, the nodes use a variant of NEWSCAST [10] to gossip among each other to exchange random values when they find that the relative order of their random values and that of their attribute values do not match. This algorithm is robust in face of frequent dynamics and guarantees a fast convergence to the same sequence of peers with respect to the random and the attribute values. At every point in time the current random value of a node serves to estimate the slice to which it belongs (its slice).

An alternative algorithm has been proposed in [5], which works by locally approximating the rank of the nodes in the ordering, without the application of random values. The basic idea is that each node periodically estimates its rank along the attribute axis depending of the attributes it has seen so far. This algorithm is robust and lightweight due to its gossip-based communication pattern: each node communicates periodically with a restricted dynamic neighborhood that guarantees connectivity and provides a continuous stream of new samples. Based on continuously aggregated information, the node can determine the slice it belongs to with a decreasing error margin. The paper shows that this algorithm provides accurate estimation at the price of a slower convergence.

4 Problem Definition

We consider a dynamic collection of nodes N, where each node i is provided with a partial function $f_i : A \rightarrow V$ associating *attribute names* taken from a domain A to *attribute values* in V.

A slice $S(c, s)$ is a dynamic subset of N defined by the following parameters:

- c is a condition expressed through first-order logic over the sets A and V (attribute names and values), identifying nodes that can be *potentially* members of the slice;
- s is the desired slice size.

The goal is to build and maintain a slice $S(c, t)$ containing only nodes satisfying c whose size approximates, as much as possible, the desired size s.

Obtaining an exact value (instead of an approximate one) is not possible in general, due to they dynamism of the system, which may prevent to reach a steady-state situation. For this reason, our figure of merit to evaluate our approach will be the *approximation quality* $q(c, s)$, defined as the ratio between the actual size and the desired one:

$$q(c, s) = \frac{|S(c, s)| - s}{s}.$$

5 The Algorithm

So far, we have discussed the slicing problem using the generic terms of "building" and "maintaining"; since we are considering only decentralized solutions, we must explain what this means from the point of view of each single node belonging to the system.

First of all, each node must be aware (i) whether it can potentially belong to the slice, because it satisfies the specified condition – in which case it is called a *potential* node; (ii) whether it currently belongs to the slice, because it has been selected among the potential nodes – in which case it is called a *member* node. Achieving (i) could be done by simply broadcasting the condition defining the slice to all nodes, while how to achieve (ii) is the subject of this section.

Second, member nodes must be aware of the membership of the slice; such information is needed by higher-level services and applications to exploit the node resources. Here, the problem is complicated by the large scale and extreme dynamism of the system. In practice, it is not possible for a node to know, at any time, the exact composition of the entire slice. Peer sampling comes to rescue here: each node obtains a sample of the entire network, while the network is kept connected by the random overlay network. These two conditions are sufficient to implement other epidemic protocols on top of peer sampling.

Third, nodes must be aware of the current number of potential and member nodes; based on this information, each node will be able to locally decide whether additional potential nodes need to join the slice, or whether existing member nodes must leave it. For this purpose, we will adopt the COUNT protocol summarized in Section 3.

Based on these considerations, we are now ready to describe the general idea behind our solution, summarized in Figure 2 and described in the following.

1. Start from the *global group* N of nodes, that includes all nodes of the system (potential or not). The global group is maintained by the lowest-level peer-sampling service.
2. Using an epidemic broadcast protocol based on random samples from the global group, inform all the nodes about the slice we want to build (by broadcasting the pair (c, s), where c is the condition and s is the size).
3. Potential nodes satisfying the condition join a second peer sampling service, whose task is to maintain the *potential group* P, i.e. the set of nodes that satisfy condition c.
4. Using a third peer-sampling service, build and maintain the slice S. At the beginning, the slice is empty.
5. Using an aggregation protocol [8], evaluate the size $size(P)$ of set P and the size $size(S)$ of set S. Such values are known to all nodes.
6. If $size(S) < s$ (i.e. the current size is less than target size), each potential node which is not already member of the slice joins the slice with probability p_{join}:

$$p_{join} = \frac{s - size(S)}{size(P) - size(S)}$$

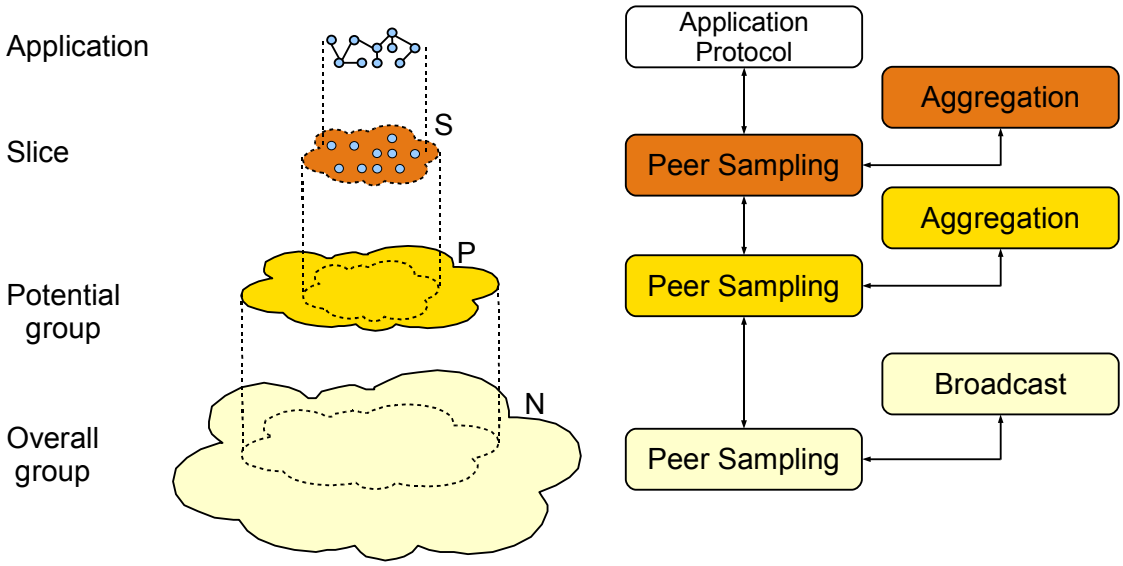

Figure 2. Architecture of the absolute slicing protocol

corresponding to the ratio between the missing nodes and the nodes available to join.

7. If $size(S) > s$ (i.e. the current size is greater than required – for example due to a variation in the user requirements), each member node leaves the slice with probability p_{leave}:

$$p_{leave} = \frac{size(S) - s}{size(S)}$$

corresponding to the ratio between the excessive nodes and the nodes currently in the slice.

8. Go to step 5.

While the general approach is simple, there are some details to be discussed.

One problem to be solved is how to *bootstrap* a new peer sampling layer from an existing one. When a new slice is created, two new peer sampling instances must be created, one containing the potential nodes, and the other containing the slice itself. The problem here is that for each group, nodes belonging to it do not know each other, yet they must populate a single, connected overlay topology.

The second problem to be solved is related to *maintenance*. After the creation of a slice, nodes belonging to it may be subject to churn, meaning that existing nodes could leave and be substituted by new ones - which are not aware of the slice, however. As a consequence, the slice would tend to continuously shrink until its death. For this reason, slices need to be continuously fed with new nodes, taken by the potential group; the potential group must be continuously fed with nodes satisfying the condition, taken by the global group.

In both the previously identified problems, the real issue is the need of nodes to discover other nodes of the same kind; i.e., potential nodes to locate other potential nodes, and member nodes to locate other member nodes. While alternative, more centralized solutions may exist, we look again for a completely decentralized approach.

We propose a modified version of the peer sampling service for maintaining the potential group and the slice. In order to enable nodes to join the potential group, all nodes in the global group participate actively (by exchanging messages) in the peer sampling layer maintaining the potential group. In order to enable potential nodes to join the slice, all nodes in the potential group participate actively (by exchanging messages) in the peer sampling layer maintaining the slice.

The differences with the original NEWSCAST protocol are the following:

- Function $getPeer()$ returns a random peer from the underlying peer sampling service, not from the local view;
- In function $prepareMessage()$, only potential nodes inject their fresh descriptors in the message they send.
- Function $update()$ works as in the original version.

These modifications have the equivalent effect of broadcasting fresh descriptors of potential nodes to all nodes in the underlying peer sampling service.

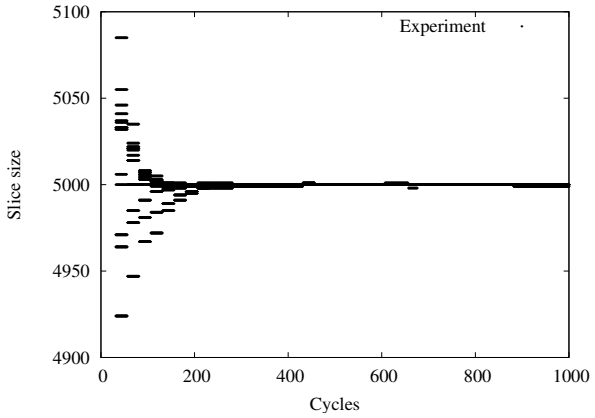

Figure 3. Static network scenario. Behavior over time of the actual slice size.

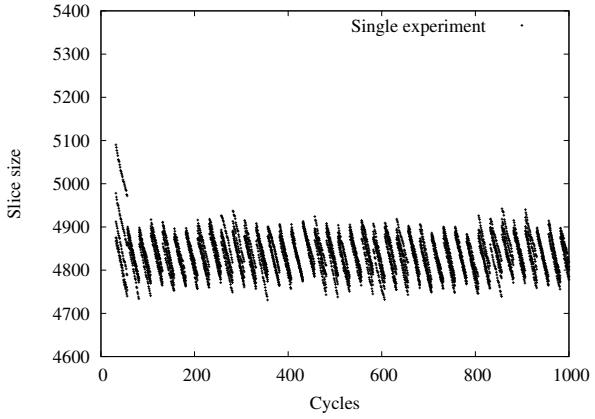

Figure 4. Dynamic network. Behavior over time of the actual slice size with churn 10^{-4}.

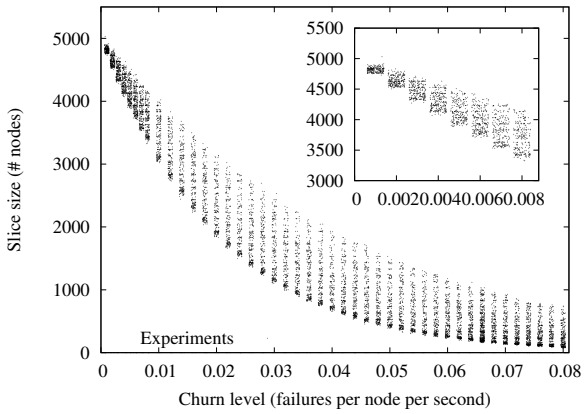

Figure 5. Dynamic network. Actual slice size with variable churn.

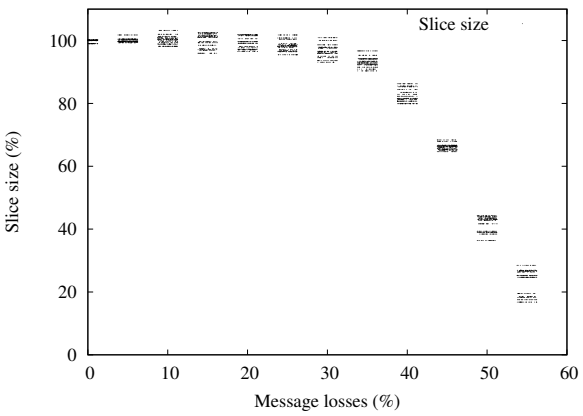

Figure 6. Static network. Actual slice size with variable message losses

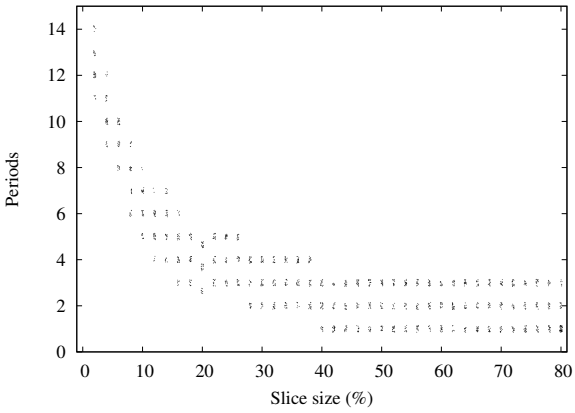

Figure 7. Static network. Time to reach convergence with variable slice size.

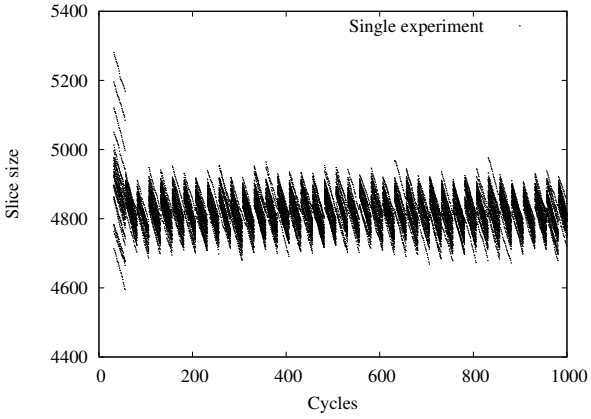

Figure 8. Realistic scenario with churn and message losses. Actual slice size.

6 Experimental Results

In this section, we experimentally evaluate the behavior of our protocol, both in the absence and in the presence of churn. All the experiments have been performed using PEERSIM, an open-source simulator designed for large-scale P2P systems and publicly available at SourceForge [9]. In all figures, 20 individual experiments were performed for all parameter settings. Whenever possible, the result of each experiment is shown with single dots to illustrate the entire distribution; in that case, the x-coordinates are shifted by a small random value so as to separate results having similar y-coordinates.

The size of the neighbor sets maintained and exchanged by the various instances of the NEWSCAST protocol is set to 30. Cycles length are equal to 10 s for all protocols; furthermore, in the aggregation protocol one epoch corresponds to 25 cycles. The size of the simulated network is equal to 25.000 nodes; we assume than 30% of them belongs to the potential set, corresponding to 7.500 of them; finally, we select 5.000 nodes out of these 7.500. Similar results can be obtained with larger sizes – this is a direct consequence of the extreme scalability of the component protocols.

In Figure 3 we show the behavior of our protocol in a static network, with no churn. The figure shows the size of the actual slice size, as measured by an external observer. It is possible to see that in the absence of failures, the target size is quickly reached. The slight oscillation that can be observed in the actual slice size are motivated by the stochastic behavior of the protocol.

In Figure 4 the behavior in case of churn is shown. As before, the figure represents the actual size of the slice. The level of churn shown in the figure corresponds to 10^{-4} leave/joins per node per second, meaning that during an epoch, 2.5% of the network nodes suddenly fail or voluntarily leave, and are substituted by the same number of new nodes (keeping the size of the network constant). Such level of churn corresponds to the typical behavior of some P2P networks [3]. It is possible to observe two important facts: the actual size of the slice is smaller than the requested size (up to 4%), and it periodically oscillates. These facts are partially motivated by the particular implementation of the decentralized aggregation protocol, as described in [8], which slightly overestimate the actuals size; and partially by the fact that during an epoch, existing nodes leaves the slice and new nodes will not join it until the epoch is restarted. This has the effect of reducing the probability for a node to join the slice, and of generating the diagonal stripes in Figure 4.

Figure 5 illustrates the behavior of the protocol with different levels of churn. It is possible to observe that at higher levels of churn correspond slices smaller and smaller. Yet, the scenarios evaluated here are extreme in their pessimism; the typical behavior described above corresponds to the leftmost group of dots in the subfigure of Figure 5.

Figure 6 illustrates the behavior of the protocol with different levels of message losses. It is possible to observe that only in very critical scenarios, message losses start to have an important impact on the quality.

Figure 7 illustrates the behavior of the protocol when you variate the size of the selected slice. This figure shows space for potential improvement of our work: building small slices is less performant than building larger slice; this is motivated by the difficulty of putting together the nodes when there a are a few of them.

of the protocol with different levels of churn. It is possible to observe that at higher levels of churn correspond slices smaller and smaller. Yet, the scenarios evaluated here are extreme in their pessimism; the typical behavior described above corresponds to the leftmost group of dots in the subfigure of Figure 5.

Finally, Figure 8 combines the previous scenarios and shows a realistic situation, where together with a churn level of 10^{-4} failures per node per second, 2% of the messages get lost.

7 Conclusions

In this paper, we have introduced an alternative version of the generic distributed slicing problem, called absolute slicing. In this problem, the goal is to assign a specified number of nodes to a slice and maintain such assignment in spite of churn. We have proposed a simple algorithm that solves the problem by combining well-known protocols such as peer sampling and aggregation, and we have experimentally evaluated its performance. While many optimizations are possible, we believe that the pedagogical value of this work lies in the composition of simple, light-weight protocols.

A potential extension of this work could be a third instance of the distributed slicing problem, called *cumulative slicing*. Given a subset $S \subseteq P$ and an attribute a, we define the *cumulative sum* $CS(S, a)$ of attribute a in S as the sum, over all nodes in S, of the values associated to attribute a:

$$CS(S, a) = \sum_{i \in S} f_i(a)$$

The goal of the cumulative slicing problem is to build and maintain a slice such that its cumulative sum for a given attribute is approximately equal to a target value. This would enable to express slice requests such as "give me enough machines to store 10 TB of data, such that each machine has at least 1 GB of available storage and a least a T1 line".

It is interesting to note that absolute slicing is a subproblem of cumulative slicing; it is sufficient to consider the

presence of a dummy attribute with value 1 at all nodes, and then use such attribute in the definition of the cumulative slice. Even more interesting, the architecture of our algorithm could be easily adapted to solve cumulative slicing; it would be sufficient to let a node become active with a probability proportional to the attribute value. Due to space and time constraints, the experimental analysis of such problem will be performed in a future paper.

Acknowledgements

Work supported by the project CASCADAS (IST-027807) funded by the FET Program of the European Commission.

References

[1] O. Babaoglu, T. Binci, M. Jelasity, and A. Montresor. Firefly-inspired heartbeat synchronization in overlay networks. In *Proceedings of the First IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2007)*, Boston, USA, July 2007.

[2] O. Babaoglu, M. Jelasity, A.-M. Kermarrec, A. Montresor, and M. van Steen. Managing clouds: A case for a fresh look at large unreliable dynamic networks. *Operating Systems Review*, 40(3):9–13, July 2006.

[3] M. Castro, M. Costa, and A. Rowstron. Performance and dependability of structured peer-to-peer overlays. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN'04)*, Washington, DC, USA, 2004. IEEE Computer Society.

[4] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing (PODC'87)*, pages 1–12, Vancouver, British Columbia, Canada, August 1987. ACM Press.

[5] A. Fernandez, V. Gramoli, E. Jimenez, A.-M. Kermarrec, and M. Raynal. Distributed slicing in dynamic systems. In *Proceedings of the 27th International Conference on Distributed Computing Systems (ICDCS '07)*, page 66, Washington, DC, USA, 2007. IEEE Computer Society.

[6] M. Jelasity and O. Babaoglu. T-Man: Gossip-based overlay topology management. In S. A. Brueckner, G. Di Marzo Serugendo, D. Hales, and F. Zambonelli, editors, *Engineering Self-Organising Systems: Third International Workshop (ESOA 2005), Revised Selected Papers*, volume 3910 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 2006.

[7] M. Jelasity and A.-M. Kermarrec. Ordered slicing of very large-scale overlay networks. In A. Montresor, A. Wierzbicki, and N. Shahmehri, editors, *Peer-to-Peer Computing*, pages 117–124. IEEE Computer Society, 2006.

[8] M. Jelasity, A. Montresor, and O. Babaoglu. Gossip-based aggregation in large dynamic networks. *ACM Trans. Comput. Syst.*, 23(1):219–252, 2005.

[9] M. Jelasity, A. Montresor, G. P. Jesi, and S. Voulgaris. PeerSim - Peer-to-Peer simulator. `http://peersim.sourceforge.net`.

[10] M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. van Steen. Gossip-based peer sampling. *ACM Transactions on Computer Systems*, 25(3):8, Aug. 2007.

[11] C. Kesselman and I. Foster. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.

[12] A. Montresor, M. Jelasity, and O. Babaoglu. Chord on demand. In *Proceedings of the 5th International Conference on Peer-to-Peer Computing (P2P 2005)*, pages 87–94, Konstanz, Germany, Aug. 2005. IEEE.

[13] Planet Lab. `http://www.planet-lab.org/`.

[14] The BOINC Project. `http://www.boinc.org/`.

[15] S. Voulgaris, A.-M. Kermarrec, L. Massoulié, and M. van Steen. Exploiting semantic proximity in peer-to-peer content searching. In *Proceedings of 10th IEEE International Workshop on Future Trends of Distributed Computing Systems (FTDCS 2004)*, pages 238–243, 2004.

[16] S. Voulgaris, E. Riviere, A. Kermarrec, and M. van Steen. Sub-2-sub: Self-organizing content-based publish and subscribe for dynamic and large scale collaborative networks. In *Proceedings of the Fifth International Workshop on Peer-to-Peer Systems (IPTPS06)*, 2006.