Coordinated Self-Adaptation in Large-Scale Peer-to-Peer Overlays

Corina Stratan Jan Sacha Jeff Napper Guillaume Pierre
Department of Computer Science
VU University Amsterdam, The Netherlands
E-Mail: {cstratan,jsacha,jnapper,gpierre}@cs.vu.nl

Technical report IR-CS-60, Vrije Universiteit, September 2010.

Abstract

Self-adaptive systems typically rely on a closed control loop which detects when the current behavior deviates too much from the optimal one, determines new optimal values for system parameters, and applies changes to the system configuration. In decentralized systems, implementing each of these steps is challenging, especially when nodes need to coordinate their local configurations. In this paper, we propose a decentralized method to automatically tune global system parameters in a coordinated manner. We use gossip-based protocols to continuously monitor system properties and to disseminate parameter updates. We show that this method applied to a decentralized resource selection service allows the system to quickly adapt to changes in workload types and node properties, and only incurs a negligible communication overhead.

Keywords: Self-adaptation, peer-to-peer, resource selection service, XtreemOS.

Contents

1 Introduction **3**

2 Related Work **4**

3 Design Overview **5**
3.1 Starting a Self-Adaptation Instance 5
3.2 Choosing New Parameter Values 6
3.3 Coordinated System Reconfiguration 6

4 Case Study: Resource Selection Service **7**
4.1 Self-Adapting the RSS 7
4.2 Monitoring 10
4.3 Optimizing 10
4.4 Reconfiguring 12

5 Evaluation **14**
5.1 Experimental Setup 14
5.2 Adaptation to Changes in Node Properties 15
5.3 Adaptation to Varying Query Workloads 18
5.4 Impact on the Query Delivery 19
5.5 Self-Adaptation Cost 22

6 Conclusions **22**

1 Introduction

Complex distributed systems usually have many internal configuration parameters. However, setting parameters to fixed values that work well in all cases is often impractical or impossible. A common technique to maintain good system behavior in the presence of changes in the run-time conditions is to control such parameters using self-adaptation. Conceptually, a self-adaptive system is very simple: The system continuously (i) detects when behavior deviates too much from optimal, (ii) derives new parameter values, and (iii) applies these changes. However, achieving these three steps in practice can be very difficult depending on the characteristics of the application.

One challenging application domain for self-adaptation is composed of large scale peer-to-peer (P2P) overlays. Such applications usually exhibit good properties of scalability and tolerance to failures. However, they also have very dynamic and complex behavior: computing resources might join or leave the system at any time; operating systems and software systems might be upgraded; and application requirements might change over time. To maintain efficient behavior in the presence of such changes, applications often need to control a number of internal parameters. Some parameters may be tuned independently at each node, such as the number of items stored by each node and the number of connections maintained with other nodes [1, 13]. However, other parameter tuning requires coordination between a vast majority of nodes to maintain application correctness.

Most of the self-optimization solutions proposed so far for peer-to-peer overlays focus on adjusting local independent parameters and ignore coordinated parameters. However, such "global" parameters are very common. For example, threshold values can be used as criteria to group nodes into smaller clusters [6]; the periodicity of certain actions is fixed (like exchanging maintenance messages); or the partitioning of a virtual Cartesian space in which the nodes are placed is shared among all nodes [5]. Currently, these settings are done by human system administrators and are not amenable to dynamic control during the system's lifetime. Long-lived systems, however, can often improve performance by adapting such global parameters to changes in the run-time conditions.

This paper addresses self-adaptation in systems that are both: (i) decentralized, where a large number of functionally identical compute nodes collaborate to realize the system's functionality; and (ii) coordinated, where the system requires that all nodes use the exact same set of parameters at any point in time. The goal is to complement existing applications with a separate control plane to cheaply, quickly, and automatically adapt their global parameters.

Addressing coordinated self-adaptation in such a context requires ensuring that a coordinated parameter retains the exact same value through the entire system. For example a DHT that allows nodes to select the optimal key size autonomously could not work correctly if a fraction of nodes selected an 'optimal' key size of 128 while some others chose 256. On the other hand, the control plane should be designed as a decentralized system so as not to jeopardize the benefits of decentralization of the application itself.

Our work addresses all three steps of coordinated self-adaptation in such decentralized, coordinated systems. In our system, any node may probabilistically elect itself as the leader to conduct the next round of adaptation. To choose new parameter values, the leader must monitor the global system behavior. On one hand, fetching monitoring information from all nodes in the system would be prohibitively expensive. On the other hand, compact aggregate functions such as the average of some attribute across the system are easy to generate but they do not carry much information about the system as a whole. We claim that estimating the statistical distribution of node attributes across the overlay represents a reasonable tradeoff between a low aggregation cost and a high expressiveness to allow for accurate parameter setting. Such distributions can be obtained efficiently in a decentralized manner [10]. Finally, the decisions are disseminated through a gossip-based protocol to ensure a fast convergence of the system to a new state. We can then provide synchronization through a simple changeover policy when nodes receive the new state.

We demonstrate self-adaptation of global parameters in the context of the Resource Selection Service (RSS) developed as part of the XtreemOS operating system for the Grid [3]. The RSS is a scalable, fully decentralized system to identify resources that match application requirements in large-scale utility computing infrastructures. Resources are organized in a multidimensional space where each dimension represents a resource attribute. To allow efficient searching, this multidimensional space is split into cells with arbitrary boundaries. Importantly, the search protocol requires that all nodes use identical cell boundary definitions. The goal of self-adaptation in this case study is to place cell boundaries in a running system such that queries return correct results while traversing the lowest possible number of nodes.

Our results demonstrate the usefulness of adapting cell boundaries: introducing coordinated self-adaptation in our case study reduces query costs by a factor of 4 compared to an initial human-selected configuration. We also demonstrate efficient self-adaptation to changes in the distribution of queries for resources and to long term changes in the distribution of the attributes of nodes themselves. Finally, we show that the system's delivery remains high even during coordinated live reconfiguration from one set of parameters to another.

The rest of this paper is organized as follows. Section 2 describes related work. Section 3 presents our coordinated self-adaptation model for decentralized systems. Section 4 describes an application of this model to the Resource Selection Service. Section 5 evaluates the self-adaptation protocol and finally Section 6 concludes the paper.

2 Related Work

To our surprise, previous work in self-adaptation in large-scale decentralized systems focuses entirely on tuning parameters at the local level rather than on "global" coordinated parameters. For example, tuning local node parameters can be used for decentralized load balancing [14], load-balancing storage and replication in DHTs [1], adapting the critical exponent of power-

law networks [12], and maintaining the optimal ratio of super-peers to sub-groups [11, 13]. Each of these is a difficult problem in itself, and one of the paramount difficulties of tuning local node parameters is to ensure that the system converges without coordination to an optimum global state. Our work instead focuses on directly adapting global parameters requiring coordination while still maintaining a completely decentralized system. To the best of our knowledge, this is the first work specifically addressing the self-adaptation of global parameters in a decentralized setting.

Our work uses a decentralized monitor for P2P systems. Some large-scale monitoring systems use hierarchical aggregation by building a tree-like topology to collect data (for example, [15,16]). However, the hierarchical topology is difficult to construct and maintain in the presence of churn, which is always present in P2P systems. A different direction builds on fully decentralized gossip-based aggregation to obtain compact statistical results like averages or total counts. This method is simple to implement, robust to churn, and provides any node in the system with the computed statistics. The compact values obtained are however not always sufficient for optimization tasks. The decentralized monitor we use in this paper provides the distribution of the values of a parameter at low cost with all the advantages of other aggregation methods [10].

3 Design Overview

We consider a system consisting of a large number of autonomous nodes that self-organize into a peer-to-peer overlay. There is no central point of control of the system; instead, nodes, or *peers*, communicate through gossip protocols to achieve a group goal. Each peer knows about a small subset of the other peers, called *neighbors*, that form an overlay. Peers periodically exchange lists of neighbors to maintain the overlay even in the presence of *churn* wherein frequently new peers join and old peers leave the system [9].

3.1 Starting a Self-Adaptation Instance

Addressing coordinated self-adaptation for scalable distributed systems requires finding a sweet spot between a fully decentralized design where each node may choose its own parameter value autonomously and a centralized one in which a single leader chooses a global parameter value and imposes it on the rest of the system. The first design faces the risk that different nodes choose different values for the same parameter, which contradicts the goal to have a uniform value across the whole system, while the centralized approach introduces a single point of failure and a potential performance bottleneck.

To perform efficient coordination our system relies on self-elected leaders, where each self-elected leader is in charge of carrying out one instance of the adaptation protocol through the whole system. At each round, each node may elect itself as a leader with a probability inversely proportional to the number of nodes in the system. This allows maintaining the average frequency at which leaders emerge from the system. However, it does not prevent multiple

adaptation instances from executing simultaneously in the system. We therefore impose a global order between protocol instances using a totally ordered timestamp so that newer instances can supersede older ones. This also allows nodes to garbage collect old instances whose leader failed before the completion of its task.

3.2 Choosing New Parameter Values

An adaptation leader needs information about the global system state to select new parameter values. Here as well, we observe a necessary tradeoff regarding the quantity of information made available to the leader. On the one hand, bringing detailed information about each peer may allow the leader to make accurate complex choices, but the costs of gathering such exhaustive information may be prohibitive, especially in large-scale distributed systems. On the other hand, distributed aggregation algorithms can efficiently compute functions such as the average of some attribute value across the system. However, a single aggregate value such as an average might not provide enough detail to optimize the global configuration. In particular, average values are very sensitive to the presence of a small number of outliers in the system.

We argue that a reasonable tradeoff consists of estimating the statistical distribution of some attribute across the system. As discussed in Section 4, a statistical distribution captures essential information about the system: it shows the full spectrum of node characteristics in the system, and the proportion in which they exist. This allows nodes to make complex decisions where a balance between multiple contradictory requirements is often involved.

Estimating the statistical distribution of some attribute across a large-scale distributed system can be realized both accurately and inexpensively. In this paper we use our own Adam2 algorithm which efficiently approximates node attribute distributions in a fully decentralized manner [10]. Using successive rounds of gossip protocols, Adam2 collects and refines attribute distribution approximations, quickly converging at all nodes to an accurate distribution estimation.

3.3 Coordinated System Reconfiguration

Relying on self-elected leaders for adaptation allows the system to take unambiguous decisions regarding the values that coordinated parameters should have. However, to achieve our goal we must also apply these reconfigurations such that all nodes receive the same new configuration and transition to the new configuration in a coordinated manner, without the need to stop the application.

New configurations are disseminated through the system using the same gossip-based protocol as in Adam2. In this manner, a single decision can be propagated consistently to all nodes in the system (probabilistically) within a few gossip cycles. Live reconfiguration of the system is however not trivial. For example, in our case study, the system needs to continuously route user queries which requires all nodes to use consistent configurations in order to avoid loops

and other routing errors. For this reason, nodes temporarily maintain multiple configurations during adaptation, and associate each query with an identifier of the configuration that should be used to process it. Outdated configurations are eventually dropped after a timeout that is longer than the expected propagation delay for a new configuration. Since configurations are spread exponentially using gossip, the propagation delay is sensitive only to the log of the system size.

4 Case Study: Resource Selection Service

In this section we describe the application of our self-adaptation framework to the Resource Selection Service (RSS) [4]. The RSS is a vital component of the XtreemOS Grid operating system: It provides the fundamental *lookup* primitive that takes a specification of resource attributes required by an application, and returns a list of machines suitable for running the concerned application. The RSS is designed according to a fully decentralized design. Compute nodes themselves are fully responsible for managing their own attributes and collaboratively answering lookup queries.

Each node n in the RSS is assigned a sequence of D static *attributes*, $A_1(n), A_2(n), \ldots A_D(n)$, which define the node's relevant properties such as the CPU architecture, operating system version, amount of installed RAM, library versions, etc. The system allows nodes to submit multidimensional range queries based on these attributes to find nodes whose attributes satisfy the query constraints. In order to quickly route these queries to the relevant nodes, all RSS nodes participate in a structured P2P overlay network generated using the D-dimensional attribute Cartesian space (see Figure 1). Node n is thus represented in the overlay network as a point with coordinates $(A_1(n), A_2(n), \ldots A_D(n))$.

Queries over ranges of attributes are represented in the same space as nodes to enable efficient routing. A query q is defined as a sequence of D ranges $((q_1^{min}, q_1^{max}), (q_2^{min}, q_2^{max}), \ldots (q_D^{min}, q_D^{max}))$ and corresponds to a hyper-rectangle (i.e., Cartesian product of intervals) in the virtual space. By this definition, all nodes that belong to the query hyper-rectangle satisfy the query requirements.

To determine neighbor connections and organize query routing in the P2P overlay, the virtual space is partitioned into $(K+1)^D$ *cells* using a multidimensional grid consisting of K *cell boundaries* in each dimension. Nodes maintain a list of neighbors that are located in a hierarchical set of cells. To handle a query, an RSS node first identifies all cells in the system that intersect with the query hyper-rectangle, and forwards the query to a node in each of these cells using a hierarchical routing protocol that is bounded by $O(D \log K)$ messages [4]. Finally, nodes in the respective cells use a simple sequential flooding protocol to route the query to the rest of the nodes in the cell.

4.1 Self-Adapting the RSS

The RSS is a very good candidate to study self-adaptation of the global configuration of a decentralized system: the query routing overhead can be sig-

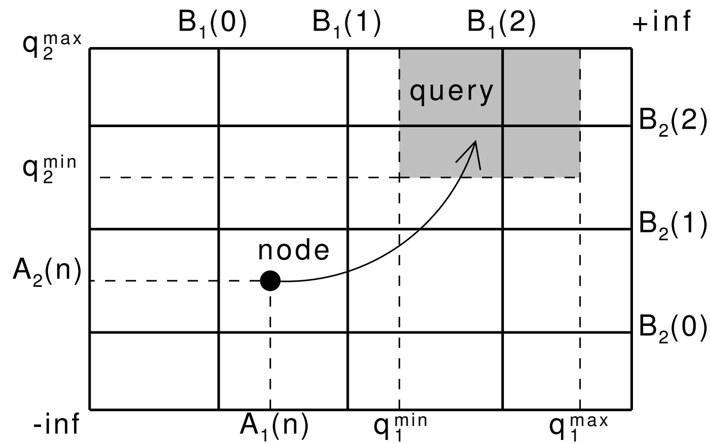


Figure 1: RSS query routing in two dimensions. Every node is placed in a multidimensional space according to its attributes, and queries are represented by hyper-rectangles of the attribute space.

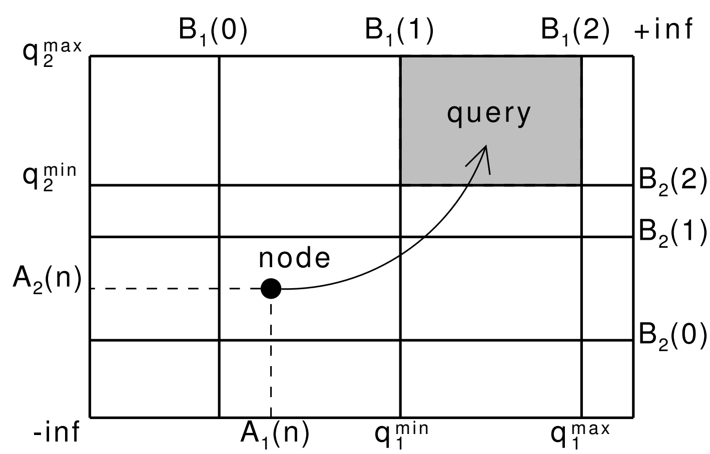


Figure 2: Optimum boundary placement for a given query forms a single cell coincident with the query.

nificantly reduced by tuning the location of cell boundaries, and changing the cell boundaries requires coordination. With respect to coordination, queries and node management in the RSS are entirely decentralized but rely on the global configuration represented by the cell boundaries in the attribute space for routing. Since the routing and query space are intertwined, if two nodes do not agree on the placement of boundaries, then they will disagree on routing. This can lead to queries not being routed to some of the matching nodes or even to routing loops. Hence, all nodes must agree on the global configuration of cell boundaries to produce correct query results.

The performance of RSS can be improved by adjusting the cell boundaries, as the query routing overhead depends directly on the placement of these boundaries. Figure 2 shows how the sample system depicted in Figure 1 could have been configured to reduce the overhead of handling the sample query. The overhead of a query consists of the costs of routing the query to all cells that overlap the query and routing among nodes within each cell. To minimize both costs, an optimal location for the boundaries for any given query are exactly the ranges of the query itself. In Figure 1, a large query overlaps four different cells that might have to be explored, parts of which do not match the query. In these cells the query can be forwarded to many nodes that might not match it. In Figure 2, the query is routed to only one cell where all nodes are matching. The performance can also be improved by adjusting the cell boundaries such that the nodes be evenly distributed across cells. This is because, as there is no structure within a cell, the routing overhead inside the cell becomes significant when there is a large number of nodes belonging to it.

The system cannot reconfigure its boundaries for each query, or whenever a node joins or leaves, because the overhead of restructuring the overlay would be prohibitive. As discussed later, we initiate reconfigurations at certain time intervals. The goal of our adaptation algorithm in the RSS is therefore to configure cell boundaries to minimize the overhead for handling queries as both the distribution of queries and nodes change over time in the system.

We modify the RSS slightly to implement our self-adaptation approach: The system should maintain query throughput during reconfiguration, requiring a live change to the global configuration. Specifically, we extend the gossip protocol used to maintain the RSS overlay, by adding reconfiguration information to the messages. We introduce the notion of a configuration consisting of a unique, totally-ordered timestamp and the cell boundaries of the multidimensional attribute space used for routing in the P2P overlay. The same timestamps are used to annotate queries so that every query is associated with the configuration containing the same timestamp. Finally, neighbor lists are also locally associated with a configuration. Every node associates a different neighbor list with each configuration that it receives (or creates) although only the list of neighbors corresponding to the most recent configuration is maintained.

The following sections describe the main steps in our adaptation algorithm: a leader gathers sufficient information to improve the boundary configuration, then calculates new boundaries if necessary, and finally disseminates the new configuration across all nodes in the running system.

4.2 Monitoring

To minimize query overhead over time with changes in both the distribution of queries and nodes, current nodes need to coordinate a reconfiguration of the cell boundaries in the RSS. Knowledge of all the previous queries and all current nodes in the system can of course be used to calculate the best expected configuration going forward. However, full global knowledge is not scalable in large systems. For this reason, we monitor instead the statistical distributions of recent queries and node attributes. As discussed later, we use the distribution of node attributes to evenly divide the number of nodes between the cells in the virtual space, and we use the query distributions to overlap cell boundaries with the most frequent query ranges.

We monitor node attribute and query range distributions using Adam2 [10], a fully decentralized aggregation protocol for the statistical distributions of values. Adam2 approximates distribution functions by estimating their values in a few carefully selected points and interpolating between these known points. Each distribution approximation is produced by a sequence of aggregation *instances* composed of a fixed number of gossip rounds. Instances iteratively refine the interpolation point placement. Adam2 is also able to tune its own approximation accuracy during the refinement process.

In the self-adaptive RSS, instances of Adam2 are continuously initiated by self-elected leaders. A leader maintains two distribution approximations for each dimension d: the node attribute distribution $Attr_d$, and the query range distribution $Query_d$. The attribute distribution is a function $Attr_d : \mathbb{R} \rightarrow \mathbb{R}$ defined such that $Attr_d(x)$ is equal to the fraction of nodes in the system that have a value for attribute A_d below x. This distribution provides the leader enough information to place cell boundaries to balance the number of nodes in each cell.

Monitoring the distribution of queries is trickier than monitoring node attributes because queries are composed of ranges in each dimension. As we discuss next, for the placement of cell boundaries, the leader is interested only in the distribution of the endpoints of query ranges. We thus define the query distribution for dimension d as a function $Query_d : \mathbb{R} \rightarrow \mathbb{R}$ such that $Query_d(x)$ is equal to the fraction of all the endpoints of query ranges (upper or lower) for dimension d. In order to reduce the influence of old queries and to reduce bookkeeping, nodes cache received queries for only T time units to form the query distribution.

4.3 Optimizing

Using the distributions of node attributes and query ranges obtained from Adam2, the leader of the instance creates a configuration consisting of a new unique, totally-ordered timestamp and (possibly new) cell boundaries. The new configuration is then installed by the leader and will then be spread by gossip as discussed shortly. Note that the leader installs a new configuration even if it differs only slightly from the previous configuration because the precise placement of cell boundaries is important to reduce query overhead.

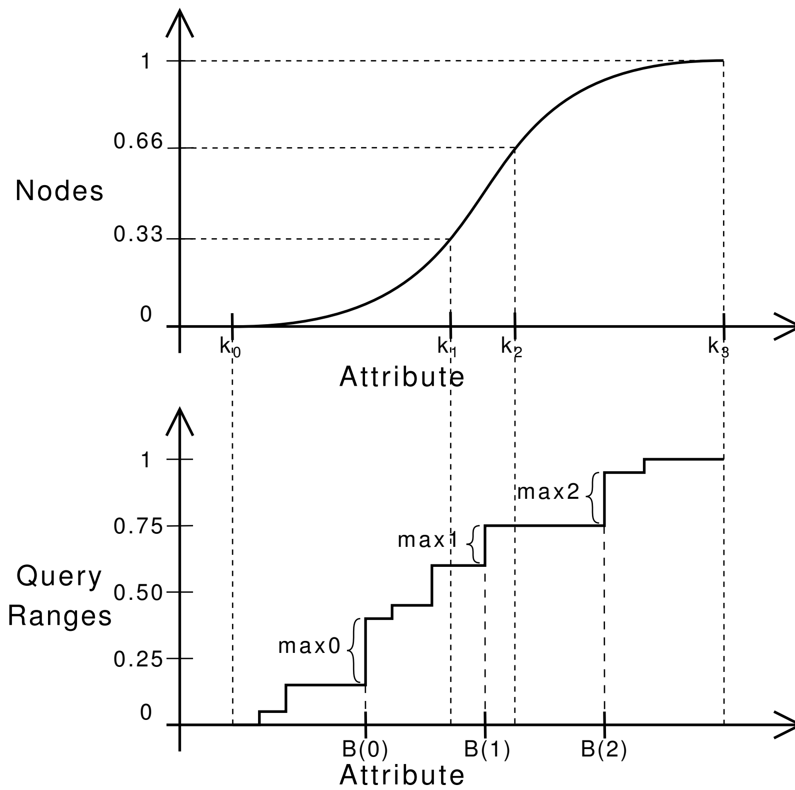


Figure 3: The boundary calculation algorithm first finds $(k_0, \ldots, k_3)$ to balance the number of nodes in each partition (above), and then places a boundary B in each partition at the largest grouping of query endpoints (below).

To create a configuration with good expected performance, the leader must solve an online optimization problem with future queries assumed to be similar to recent past queries. As discussed previously, the optimal placement of boundaries for a particular query creates a cell precisely the same size as the query to minimize routing costs to only one cell and within the cell all nodes match the query. A larger cell than the query runs the risk of routing to some nodes within the cell that do not match the query, while smaller cells than the query result in higher routing overhead between all the cells that overlap the query. The tradeoffs in cell size are complex and interdependent.

We use two heuristics to simplify the calculation of cell boundaries: (i) we attempt to balance the number of nodes in each cell to prevent the formation of overly large or small cells; and (ii) we attempt to place the boundaries of the cells coincident with the endpoint of query ranges. Specifically, the cell boundary algorithm calculates a new set of K cell boundaries $B_d(0), B_d(1), ...B_d(K-1)$ for each dimension d independently. The calculation follows our two heuristics in order as shown in Figure 3. In the first step, the node attribute distribution is used to define initial intervals for boundary values such that the nodes are roughly balanced between all cells. Specifically, $K+1$ points are calculated, $k_0, k_1, ...k_K$, such that $Attr_d(k_i) = \frac{i}{K}$. The calculation of k_i points is straightforward since $Attr_d$ is a non-decreasing function approximated by line segments, which can be easily inverted. The final value for each $B_d(i)$ boundary is later chosen such that $k_i \leq B_d(i) < k_{i+1}$ so that the cell determined by any two consecutive boundaries $B_d(i)$ and $B_d(i+1)$ contains at most $\frac{2}{K}$ of all nodes.

In the second step, precise cell boundaries are calculated using the query distribution. The goal of this phase is to place cell boundaries specifically at the most frequent query range endpoints. We choose the simple location of the most frequent endpoints instead of using more complex clustering of endpoints not for simplicity. The benefits of reducing routing overhead only appear if the cell boundaries are exactly those of the query: A small overlap of the query with another cell can force the query to be routed to all nodes in the other cell. Hence, precise placement of cell boundaries is very important to reducing routing overhead in the RSS.

The most frequent range endpoints are easily identified by the point of the largest change in $Query_d$. The height of each change in the query distribution function is by definition equal to the frequency of the corresponding query endpoint. Boundary $B_d(i)$ is placed at the greatest change in the query distribution between points k_i and k_{i+1} as shown in the bottom of Figure 3. If $Query_d$ does not change between points k_i and k_{i+1}, boundary $B_d(i)$ is simply defined as $\frac{k_i + k_{i+1}}{2}$. By aligning cell boundaries with common query ranges we reduce both the hierarchical routing overhead since queries intersect with fewer cells, and the intra-cell routing overhead since fewer non-matching nodes have to be visited when exploring partially overlapping cells.

4.4 Reconfiguring

Every node has a current configuration containing cell boundaries and an associated timestamp, and an associated list of neighbors based on the configuration's

set of boundaries. When a node first joins the P2P system, it starts with a default configuration. If a node installs a new configuration, it associates a new neighbor list corresponding to the new cell boundaries.

In order to spread the new set of boundaries from the leader to the rest of the system, we extend the periodic gossip messages used to maintain neighbor lists in the overlay. During this gossip, nodes exchange their current timestamps. If a node discovers that its neighbor has a higher timestamp, it requests the corresponding configuration from this neighbor, installs the new configuration, and creates a new, local corresponding neighbor list. Thus, each new configuration spreads epidemically between the nodes, quickly reaching all nodes even in very large systems.

The configuration propagation protocol is fast and efficient, but it can cause temporary inconsistencies between nodes with different configurations. In order to enable live system reconfiguration without losing queries, nodes cache and reuse configurations. When a node receives a new configuration, it caches old configurations along with their corresponding neighbor lists. The cached neighbor lists are *not maintained*; only the most recent neighbor list is actively maintained. Overlay maintenance overhead therefore does not increase when nodes cache different configurations. The cached configurations and neighbor lists are eventually discarded after a timeout.

In order to handle a query correctly, all nodes involved in query processing must use the same configuration. For this reason, when a query is generated by a node, it is associated with the timestamp of the node's current configuration. However, to allow time for new neighbor lists to be generated, queries still use the previous configuration's timestamp for a short period after a new configuration is installed. When a node receives a query, it checks to see whether the timestamp corresponds to the current or one of the cached configurations held by the node. If a configuration is found, the node can process the query using the standard RSS protocol. If the configuration is cached, the neighbor list can be outdated, and the query might return incomplete results. If there is no corresponding configuration found, the node requests the configuration from the node that it received the query from. When the node receives the configuration, it starts building a new list of neighbors; in most of the cases, when the shifts in the boundaries are not significant, many of the previous neighbors can be reused for the new list. Finally, the query can return incomplete results as with cached configurations due to using outdated neighbor lists.

We discuss briefly concerns about multiple, simultaneous leaders. Although they might generate multiple different new configurations, the total order on timestamps will ensure only one becomes the latest configuration across the system as only the latest configuration is disseminated. However, in the interim, queries created at nodes using the slightly older configurations can begin to propagate. These queries can experience higher latency due to the need to propagate the configuration along with the query and can return incomplete results as discussed above when configurations received with the query are used to process a query.

5 Evaluation

The evaluation of our self-adaptation protocol aims to estimate the performance improvement that this protocol brings to the RSS. We evaluate our protocol in PeerSim, a simulator for peer-to-peer systems [8] which allows us to model systems with large numbers of nodes. We compare two versions of the RSS: one which uses our self-adaptation protocol, and one which does not.

We consider two test cases in our evaluation: an adaptation to changes in the node population, and an adaptation to changes in query workloads. In both test cases, we simulate two types of changes: sudden changes (e.g., an addition of a new computing cluster to the system, or a switch from one type of application to another), and gradual changes (e.g., a system in which old machines are gradually replaced by new machines, or a slow transition in the type of jobs that users tend to run in the system).

We assess the improvement in the RSS performance by measuring the RSS routing overhead, defined as the average number of nodes traversed by a query. This metric captures both the query routing cost and query latency, since RSS queries traverse nodes sequentially. We also investigate the impact of our self-adaptation protocol on the RSS responsiveness by measuring the query delivery rate, defined as the average fraction of nodes correctly discovered by a query. Finally, we measure the extra maintenance cost introduced in the RSS by our self-adaptation protocol.

5.1 Experimental Setup

Although we evaluate our system through simulation, we use real-world data to initialize node attribute values and several types of queries that closely resemble the workloads from current Grid systems. Specifically, we obtained descriptions of over 300,000 machines that participated in the the BOINC volunteer computing project between 2004 and 2008 [2]. Based on these machine descriptions, we initialize the following four node attributes in the RSS: measured CPU performance in FLOPS, measured downstream bandwidth, amount of installed memory, and amount of installed disk space.

We exercise the system with several types of synthetic query workloads that have similar characteristics to the workloads observed in real Grid systems. Although a number of job traces from Grid systems are available [7], we could not use them directly in our experiments because they mostly contain information about job runtime characteristics (e.g., total running time, amount of used memory) and give very little information about node characteristics required for job execution. In our experiments, we use the following three workload types:

- **bag-of-tasks:** a workload in which a few specific queries appear very frequently. This corresponds to the "bag-of-tasks" type of jobs, that contain a large number of very similar tasks (and thus, a large number of identical job submission queries).

- **coarse-grained:** a workload which simulates user-generated queries. In such queries, attribute ranges are specified in course-grained units. For example, the amount of RAM is specified in multiples of 512 MB.

- **random:** a workload in which all the queries specify random intervals for attribute values. We use this workload as a base for comparison with the other workloads.

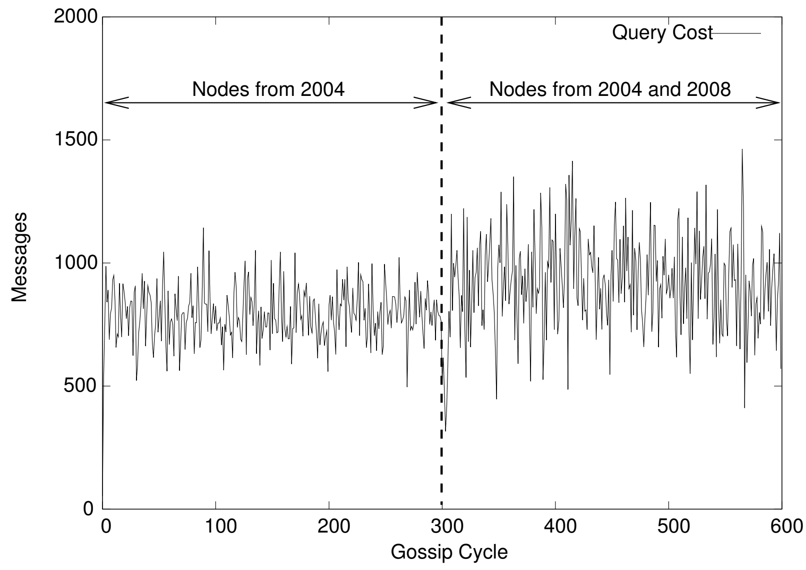
5.2 Adaptation to Changes in Node Properties

The statistical distribution of node properties may change dramatically when new machines are added to the system, or when they replace older ones. To simulate such situations, we use two sets of node properties based on the BOINC traces from years 2004 and 2008. For this particular experiment we replaced one node attribute (available downstream bandwidth) with the installed kernel version: this attribute suffers much more changes across the years, and allows to stress our system better.

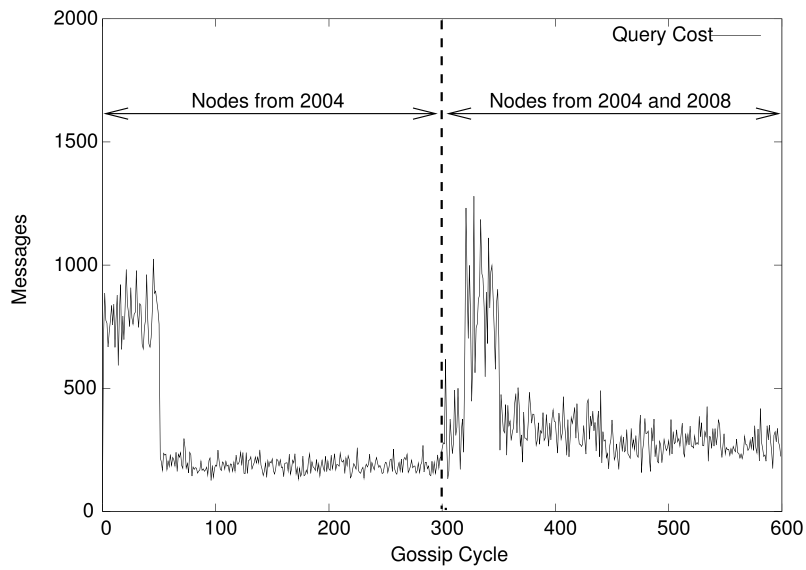
The case when a new computing cluster is added to the system creates a sudden change in the statistical distribution of node properties. We simulated this case by starting the RSS with 5,000 nodes with attribute values obtained from the 2004 traces. After 300 gossip cycles, we added 5,000 more nodes with attribute values from the 2008 trace. The query routing overhead, with and without the self-adaptation protocol running, is shown in Figure 4.

The first part of Figures 4(a) and 4(b) show the effect of self-configuration in the RSS. Both systems start with the same set of query boundaries chosen by the human operator, and experience a query cost in the order of 800 messages per query. In the adaptive system, these costs drop by a factor 4 after the first system reconfiguration. At time 300, both systems see a cost increase. Part of this increase is due to the fact that the size of the system is doubled, and therefore the number of nodes matching the queries also roughly doubles. The adaptive system also sees an additional cost increase due to the fact that its configuration is suddenly ill-suited to the workload. It however quickly adapts to this new situation and returns to an average cost four times lower than the non-adaptive system.

Figure 5 presents similar simulation results for a situation in which the node properties gradually change from one distribution to another. We create this change by starting with 5,000 nodes from the 2004 BOINC trace, and subsequently replacing a few nodes at each gossip cycle with new ones drawn from the 2008 trace. Again, the adaptive system shows much better performance than the non-adaptive one. The non-adaptive system sees a relative performance improvement until cycle 250. This is explained by the fact that, in that phase of the experiment, there is a balance between the number of old and new machines, and the nodes are distributed more evenly into cells. The adaptive system, on the other hand, issues several relatively minor reconfigurations, and maintains a constant performance despite the workload variations.

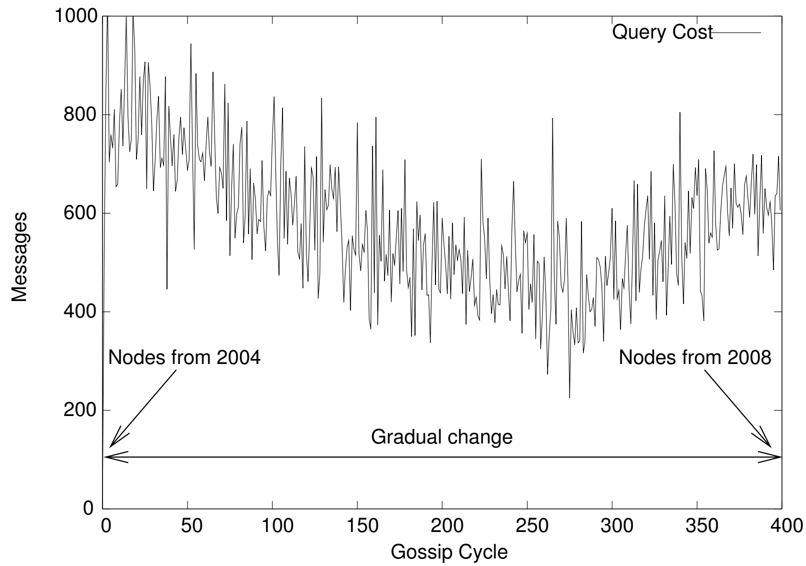


(a) Without self-adaptation

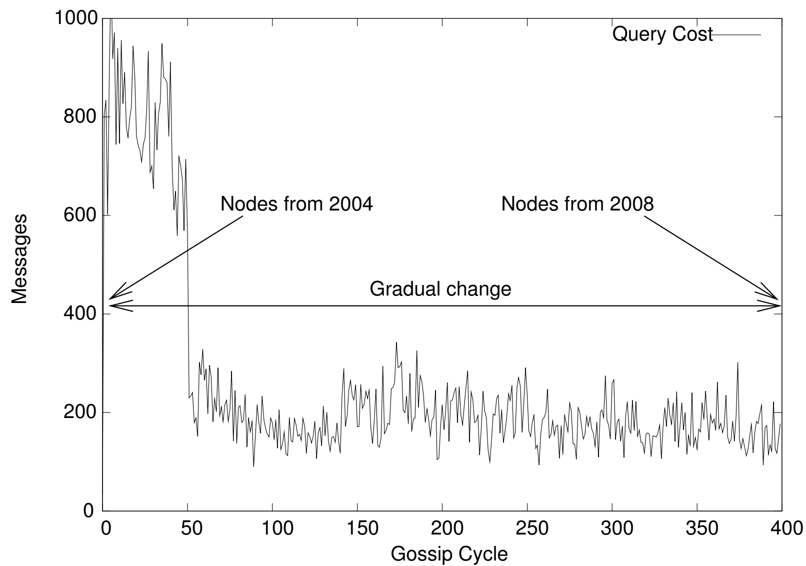


(b) With self-adaptation

Figure 4: Routing overhead for a sudden change in the node properties.

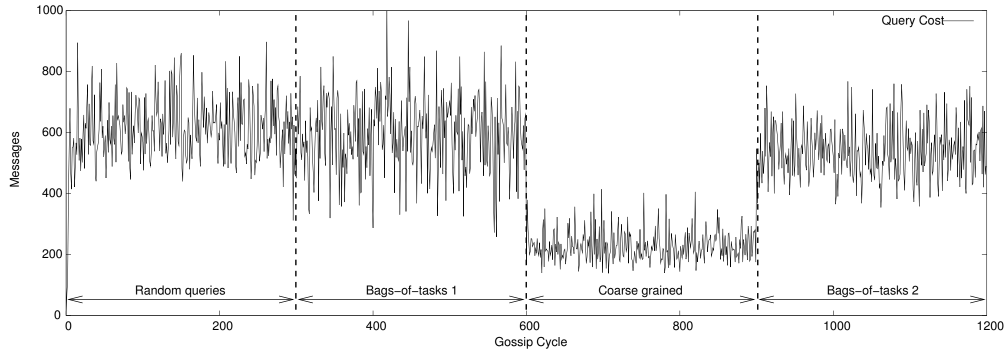


(a) Without self-adaptation

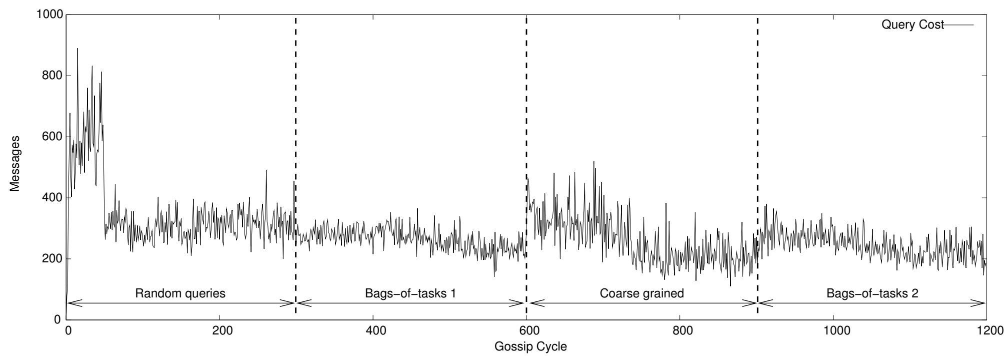


(b) With self-adaptation

Figure 5: Routing overhead for gradual changes in the statistical distribution of node properties.



(a) Without self-adaptation



(b) With self-adaptation

Figure 6: Routing overhead for sudden changes in query workloads.

5.3 Adaptation to Varying Query Workloads

We now evaluate RSS adaptation to variations in the query workloads it receives. We simulated 10,000 nodes, with attributes drawn from the 2008 BOINC trace.

We first consider sudden workload changes, by switching the query workload from one type to another at some point of time. We start the experiment with random queries, then switch to bags-of-tasks (where three frequent queries account for 25% of the workload each, and the last 25% of queries are random). We then switch to a coarse-grained workload, and finally another bags-of-tasks workload (similar to the first one, but with a different set of frequent queries).

Figure 6 shows the performance of the RSS in the adaptive and non-adaptive cases. The non-adaptive system observes no significant cost difference between workloads, except for the coarse-grained workload. This workload can in fact be considered as a best case for the manual configuration of the system, since the query ranges are aligned to the same values as the cell boundaries.

We can observe that here as well the self-adaptation protocol brings a significant cost improvement. When the workload changes at gossip cycle 600 and 900, we see a small cost increase due to the fact that the previous configuration does not work best with the new workload. However, the costs quickly decrease again thanks to self-adaptation. In particular, for the coarse-grained workload,

we can see that the self-adaptation algorithm finds a configuration very close to the manually-configured "optimal" one from the non-adaptive system.

In order to evaluate the system's behavior for a (more realistic) gradual change of workload, we model a slow transition from the coarse-grained workload to a bag-of-tasks. Figure 7 shows the results of this experiment. In the first 100 gossip cycles, all the queries submitted to the system are coarse-grained. Then, we introduce bag-of-tasks queries with an increasing frequency besides the coarse-grained queries, until the last 100 gossip cycles when all the queries are bag-of-tasks. At the beginning of the experiment both systems use the same "optimal" set of boundaries so their performance is similar. When the workload starts to change, however, the non-adaptive system sees its costs increase twofold while the adaptive system efficiently controls reconfigurations and maintains a constant performance.

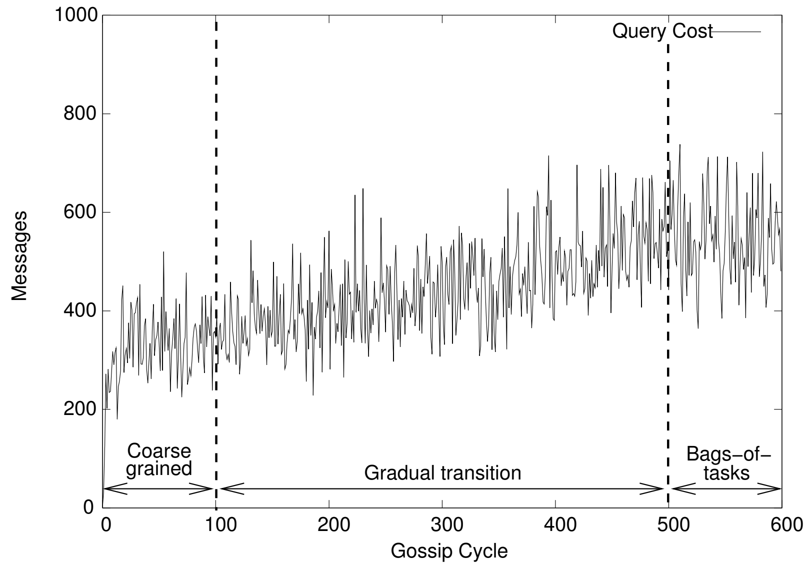
5.4 Impact on the Query Delivery

We now evaluate the impact of a runtime reconfiguration on the query delivery – that is, the number of nodes found by the RSS divided by the total number of nodes that actually match the query.

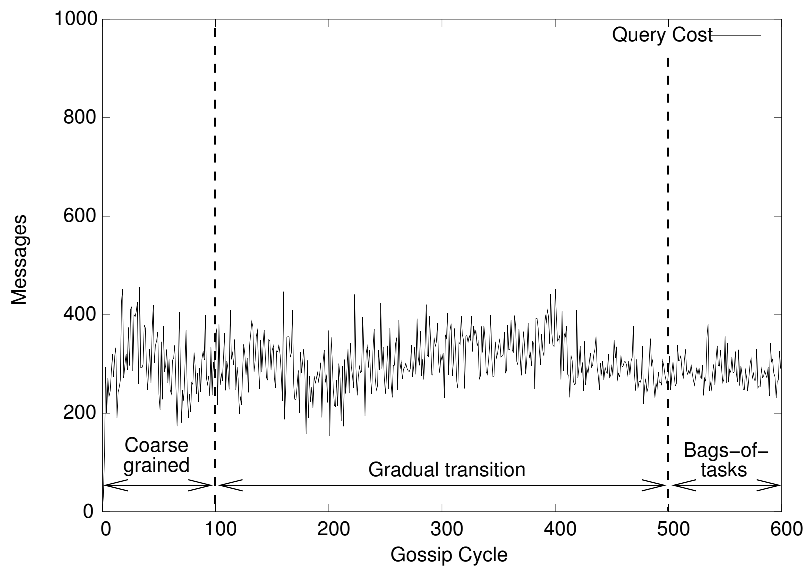
When the system starts, it takes 100 to 200 gossip cycles for each node to build a full set of neighbors. In a system with no churn nor runtime reconfiguration, the query delivery converges to 100%. When a reconfiguration occurs, each node needs to rebuild a new list of neighbors according to the new cell boundaries. However, when the reconfiguration is small, most of the previous neighbors can be reused in the new list. Only very few neighbors need to be found anew.

Reconfigurations have a second type of impact on query delivery: once a query is submitted to the system the routing algorithm assumes that all nodes use a single consistent set of cell boundaries. When a node receives a query that refers to an old set of boundaries that it does not maintain any more, all it can do is terminate the query, leading to poor query delivery.

Figure 8 shows the query delivery during the same experiment as in Figure 7: the workload gradually changes from coarse-grained queries to bags-of tasks. We show two cases: one in which each node immediately forgets its previous configuration when it receives a new one, and the case where nodes maintain a read-only cache of recent configurations. When previous configurations are not cached, the system experiences a large drop in query delivery at each adaptation. This is due to the fact that most queries present in the system at the time of reconfiguration will be terminated prematurely due to configuration inconsistencies. Figure 8(b) shows that this effect disappears when using the caching policy. In this case, delivery decreases only at the times of major reconfigurations when nodes need to seek for new neighbors. In all cases, even during reconfiguration, delivery remains high, which should remain sufficient for ensuring continuous service of the RSS within the computing grid.

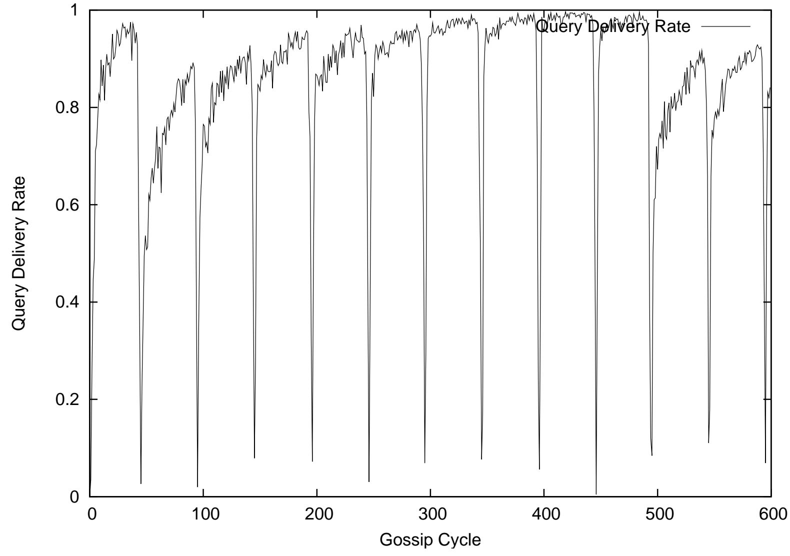


(a) Without self-adaptation



(b) With self-adaptation

Figure 7: Routing overhead for a gradual change in the query workload.



(a)

(b)

Figure 8: Query delivery rate, without (a) and with (b) caching older configurations.

5.5 Self-Adaptation Cost

An important goal of the adaptation algorithm is to incur only a small cost overhead compared to the system that is optimizing. The most important part of this overhead is the protocol's communication cost, which we estimate as follows.

The two main protocol phases that involve communication among nodes are the attribute CDF estimation through the Adam2 protocol and the dissemination of new boundary sets. As shown in [10], an Adam2 aggregation instance with 25 gossip rounds typically requires sending and receiving 40 kB of data per attribute at each node. If we consider a periodicity of one round per second, during the aggregation phase of one attribute distribution each node would need an average upstream bandwidth of 1.6 kB/s for each attribute, and a similar average downstream bandwidth. For an overlay with 4 attributes, as the one used in our tests, the needed bandwidth during aggregation is 12.8 kB/s for each node. The dissemination of new boundary sets has a significantly lower communication overhead. In order to decide whether it is necessary to reconfigure the boundary sets, the nodes periodically exchange their current timestamps of the sets. This information can be added to the regular gossip messages used to maintain the overlay, increasing their size with only 4 B. When a new boundary set is issued, each node receives it only once; for one attribute, the size of the set is normally less than 150 B.

According to our evaluations, 3 or 4 aggregation instances are usually sufficient to generate an accurate distribution approximation. Taking into account the time needed to propagate the new boundary sets after they are calculated, it takes from 100 to 200 gossip cycles to effectively reduce the routing overhead after a change in the system. If we start a gossip cycle per second, the system will be properly reconfigured within less than 200 seconds. In case such a fast reconfiguration is not necessary, gossip cycles can be initiated less frequently, resulting in a lower bandwidth consumption at the nodes.

6 Conclusions

This paper addresses coordinated self-adaptation in a large, decentralized system. It introduces a protocol that allows live adaptation of the global configuration shared by all nodes in a decentralized system in spite of the need for all the nodes to maintain configuration consistency. We demonstrate the use of this self-adaptation protocol in the Resource Selection Service (RSS), a peer-to-peer system for resource node discovery using multidimensional range queries. Our protocol allows the RSS to adapt its configuration to both gradual and abrupt changes in node properties and query workloads, decreasing the routing overhead up to four times and only marginally reducing the query delivery rate during configuration transitions. Even though we present only one case study, we believe our self-adaptation protocol is generic and can be successfully applied to other large-scale decentralized systems that use statically configured or hard-coded global settings.

References

[1] Karl Aberer, Anwitaman Datta, and Manfred Hauswirth. Multifaceted simultaneous load balancing in DHT-based P2P systems: A new game with old balls and bins. In *Self-star Properties in Complex Information Systems, LNCS 3460*, pages 373–391, 2005.

[2] David P. Anderson and Kevin Reed. Celebrating diversity in volunteer computing. In *Proc. HICSS*, pages 1–8, January 2009.

[3] Massimo Coppola, Yvon Jégou, Brian Matthews, Christine Morin, Luis Pablo Prieto, Óscar David Sánchez, Erica Yang, and Haiyan Yu. Virtual organization support within a grid-wide operating system. *IEEE Internet Computing*, 12(2), 2008.

[4] Paolo Costa, Jeff Napper, Guillaume Pierre, and Maarten van Steen. Autonomous resource selection for decentralized utility computing. In *Proc. ICDCS*, 2009.

[5] Kaoutar Elkhiyaoui, Daishi Kato, Kazuo Kunieda, Keiji Yamada, and Pietro Michiardi. A scalable interest-oriented peer-to-peer pub/sub network. In *Proc. P2P*, pages 204–211, 2009.

[6] Michael J. Freedman, Eric Freudenthal, and David Mazières. Democratizing content publication with coral. In *NSDI*, pages 239–252. USENIX, 2004.

[7] Alexandru Iosup, Hui Li, Mathieu Jan, Shanny Anoep, Catalin Dumitrescu, Lex Wolters, and Dick H. J. Epema. The grid workloads archive. *Future Generation Computer Systems*, 24(7):672–686, 2008.

[8] Márk Jelasity, Alberto Montresor, Gian Paolo Jesi, and Spyros Voulgaris. The Peersim simulator. `http://peersim.sf.net`.

[9] Márk Jelasity, Spyros Voulgaris, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten van Steen. Gossip-based peer sampling. *ACM Transactions on Computer Systems*, 25(3), 2007.

[10] Jan Sacha, Jeff Napper, Corina Stratan, and Guillaume Pierre. Adam2: Reliable distribution estimation in decentralised environments. In *Proc. ICDCS*, June 2010.

[11] Marc Sanchez-Artigas, Pedro Garcia-Lopez, and Antonio F. Gomez Skarmeta. On the feasibility of dynamic superpeer ratio maintenance. *Proc. P2P*, pages 333–342, 2008.

[12] Ingo Scholtes, Jean Botev, Alexander Hohfeld, Hermann Schloss, and Markus Esch. Awareness-driven phase transitions in very large scale distributed systems. *Proc. SASO*, pages 25–34, 2008.

[13] Paul L. Snyder, Rachel Greenstadt, and Giuseppe Valetto. Myconet: A fungi-inspired model for superpeer-based peer-to-peer overlay topologies. *Proc. SASO*, pages 40–50, 2009.

[14] Tyler Steele, Vivek Vishnumurthy, and Paul Francis. A parameter-free load balancing mechanism for p2p networks. In *Proc. IPTPS*, June 2008.

[15] Robbert van Renesse, Kenneth P. Birman, and Werner Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Transactions on Computer Systems*, 21(2):164–206, May 2003.

[16] Praveen Yalagandula and Michael Dahlin. A scalable distributed information management system. In *Proc. SIGCOMM*, pages 379–390, 2004.