A Self-Organized Semantic Storage Service

Hannes Mühleisen
Freie Universität Berlin
Networked Information Systems (NBI)
Königin-Luise-Str. 24/26
14195 Berlin, Germany
muehleis@inf.fu-berlin.de

Anne Augustin
Freie Universität Berlin
Networked Information Systems (NBI)
Königin-Luise-Str. 24/26
14195 Berlin, Germany
aaugusti@inf.fu-berlin.de

Tilman Walther
Freie Universität Berlin
Networked Information Systems (NBI)
Königin-Luise-Str. 24/26
14195 Berlin, Germany
tilman.walther@fu-berlin.de

Marko Harasic
Freie Universität Berlin
Networked Information Systems (NBI)
Königin-Luise-Str. 24/26
14195 Berlin, Germany
harasic@inf.fu-berlin.de

Kia Teymourian
Freie Universität Berlin
Networked Information Systems (NBI)
Königin-Luise-Str. 24/26
14195 Berlin, Germany
kia@inf.fu-berlin.de

Robert Tolksdorf
Freie Universität Berlin
Networked Information Systems (NBI)
Königin-Luise-Str. 24/26
14195 Berlin, Germany
tolk@ag-nbi.de

ABSTRACT

Traditional approaches for data storage and analysis are facing their limits when handling the enormous data amounts of today's applications. We believe that a radical departure from contemporary architectures of stores is necessary to satisfy that central scalability requirement. One of the most promising new schools of thought in system design are swarm intelligent and swarm-based approaches for data distribution and organization. In this paper, we describe our current work on a swarm-based storage service for Semantic Web data. This storage service utilizes algorithms discovered in the behavior of ant colonies. We describe these algorithms and our enhancements to them as well as our evaluation of the implementation.

Categories and Subject Descriptors

H.3.2 [**Information Systems**]: Information Storage

Keywords

Nature-Inspired Systems, Self-Organization, Semantic Storage, Swarm intelligence

General Terms

Algorithms, Reliability

1. MOTIVATION

Semantic applications that are recently built as prototypes in the areas of e-health, e-recruitment, e-tourism or e-commerce all rely on a central semantic store. They share the need for that store to be an efficient and scalable service which can handle huge amounts of semantic data. The performance and scalability level of these storage services will be defined by use case scenarios, e.g. the future Semantic Web applications need to scale to the size of the Web and the Internet network, respectively. To illustrate the need for large-scale storage systems, the DBpedia project currently handles approx. half a billion statements [10], which is just one order of magnitude below the amount current systems are able to handle [15].

While centralized storage solutions are not scalable to Web size, other conventional approaches for distributed storage services are facing complex problems in scaling and their adaptivity to changes in network infrastructure, both requirements for large-scale semantic applications. Storage nodes can no longer be expected to be always available nor being under the control of a single organization. Thus new concepts and architectures for distributed storage have to be developed, and a more radical departure from contemporary architectures of storage might be the key for the realization of scalable storage systems. One of the most promising approaches for data distribution are swarm-based approaches. Because of the high decentralization and self-organization of swarm-based methods they are certainly promising candidates to achieve the scalability we require.

The remainder of this paper is structured as follows: First, in section 2 we present the state of the art on semantic storage services, discuss their differences and introduce one of the alternative methods for building such systems. Our vision of a Self-organized Semantic Storage Service (S4) is described in section 3. Third, in section 4 and 5 we present our implementation and a preliminary evaluation. We conclude the paper with an overall summary and a brief outlook on future work on the way to make S4 a reality.

2. RELATED WORK

Semantic storage services are a special form of storage systems for Semantic Web metadata. The recommended Resource Description Framework (RDF) [12] provides a general, flexible approach to structure and express arbitrary information: Information is represented as a directed graph, which consists of small chunks of knowledge – referred to as "statements". Statements are modeled after simple sentences in human language: They consist of subject (S), predicate (P), and object (O), all of which are represented in a triple (S,P,O).

Semantic storage systems can be categorized by their support for data storage on more than one physical computer: Central storage systems store data on one single computer, while distributed storage systems are able to store and query data on many different computers (here referred to as *nodes*). Examples for central storage systems include Jena TDB [3] and OWLIM [11], they have been shown to be able to store and perform reasoning on graphs containing twelve billion triples on "standard" computer hardware [15].

While some central storage systems support replication of their stored data, they rely on a central instance orchestrating the execution of storage and query requests. Distributed storage systems on the other hand should not rely on central nodes, as they pose single points of failure. In order to achieve this, distributed semantic storage systems such as Edutella [14], RDFPeers [2], S-RDF [18], GridVine [5] or YARS [9] have been proposed. They make use of Peer-to-Peer (P2P) technology to create an overlay network, store semantic information in a distributed way, and are able to execute queries in a cooperative fashion.

In recent years, the study of self-organizing systems has been extended to include concepts derived from behavior found in insect swarms, schools of fish, or flocks of animals [1]. The main advantage of swarm-based algorithms is their lack of central control, of hierarchies and of complex communications, while still being able to perform difficult tasks. This makes them conceptually suited for the design of distributed and scalable systems, as communications inside those systems have to be as efficient as possible. Swarm algorithms already have been applied to problems very similar to graph or triple storage.

Important examples for nature-inspired algorithms are the "foraging" and "brood sorting" principles found in ants [1]. Foraging is the process which is used by ants to bring food to their nest: They leave the nest on random paths, if they encounter food, they carry it back leaving a distinctive scent, called trail pheromones, on the way. Other ants searching for food can now pick up this scent and also locate food by following the pheromones trail away from the nest. Those scents evaporate over time, so if the food source runs out, and the scent is no longer spread, ants stop following that path. Brood sorting on the other hand is used within the nest: Ant workers cluster their brood after its development state by picking up larvae not similar to those around it and set it down, if they notice similar larvae in the area.

In SwarmLinda [13], a distributed Linda-System based on swarm algorithms, was proposed. Linda is a communication model [7] in which processes communicate via a global tuple space. In SwarmLinda this tuple space is distributed to a network of server nodes. The different operations are realized by using swarm algorithms to reach a high level of scalability and adaptivity to network changes which are both important properties for open distributed systems. SwarmLinda clusters tuples that match the same template and trails of virtual pheromones are left in the system to make these clusters traceable. Our SwarmLinda implementation has been extended in [16] adopting ant colony algorithms to realize a distributed storage for RDF triples. Both approaches aimed at clustering semantically related RDF triples.

3. SELF-ORGANIZED SEMANTIC STORAGE SERVICE

Our concept is to build a Self-Organized Semantic Storage Service (S4) which uses swarm-based algorithms to store semantic data into diverse clusters potentially spanning multiple nodes. This structures can later be exploited for efficient data retrieval.

We have adapted the SwarmLinda concept and its basic algorithms in order to enable it to store Semantic Web meta data in the RDF format. Consistent with SwarmLinda, virtual ants move over a landscape consisting of a number of nodes (servers) which are interconnected. The adapted algorithms are described in the following sections.

One of the basic requirements of the ant algorithms is a similarity metric used to calculate the relative similarity between triples. Previous approaches have for example either employed string-based distance measures or more complex metrics based on ontologies. Any metric is required to yield a relative value for any pair of triples, thus making data organization feasible. However, our approach was deliberately designed to be independent of a specific metric, so this specific challenge is not detailed further.

3.1 Algorithms for Storage and Retrieval

S4 offers two basic operations, read, write. These operations represent the basic services offered to higher system layers, which are then able to add further services such as transactions or security. An additional move operation for data maintenance is also described. The algorithms are presented in pseudo-code format, and function names not explained in detail should be regarded as behaving according to their name. Ants move between the nodes the storage network consists of to fulfill their respective tasks. Every node maintains a list of other known nodes, that list is by no means required to be complete. The ants also keep track of the nodes they have visited during their trip through the network.

Clients connect to an arbitrary node and issue their read and write requests there. The virtual ants are then spawned, handle the request, and return potential results to the node they originated from, who in turn notifies the client about the outcome of his requests.

Algorithm 1 Write operation – basic algorithm

```
Require: Triple to store $T$, index $i$, hop count $h$, drop limit $l_d$
 1: while $h > 0$ do
 2:     $N = N \cup \{currentNodeId\}$
 3:     $p_d = calcDropProbability(T_i, h)$
 4:     if $p_d > l_d$ then
 5:         $storeTriple(T)$
 6:         $spreadScentAndReturn(T_i, N)$
 7:     else
 8:         $nextNode = selectNextNode(T_i)$
 9:         $moveTo(nextNode)$
10:         $h = h - 1$
11:     end if
12: end while
13: $storeTriple(T)$
14: $spreadScentAndReturn(T_i, N)$
```

Write operation: Algorithm 1 handles write operations in S4: For each triple to be stored in the storage system, three ants are spawned, they each carry a triple to be stored and the index of the component clustering should be performed [17]. This creates three different indices, one for each component of the triple, enabling efficient lookup for triple queries containing one fixed value and two wildcards. Ants move from node to node, and on each visit they calculate the probability of the triple being stored on that very node. The calculation of this probability is not described in detail, but the amount of similar triples stored on this node, the current system load, the amount of hops left and a random factor are to be considered following the ant brood sorting algorithm as described in [1]. If a triple is stored, the ant returns to the node it originated from using its memory of the node it has visited. On the way a "scent" for the triple just stored is spread, in order to enable efficient triple retrieval as described below. If the ant decides to move to another node, it selects the node to move to by comparing the scents present on the paths to these nodes with the triple carried. It then selects the node with the best match, again a random factor is involved. If the ant runs out of hops enabling it to move, it stores the triple on the current node and returns to make sure the triple gets stored at all.

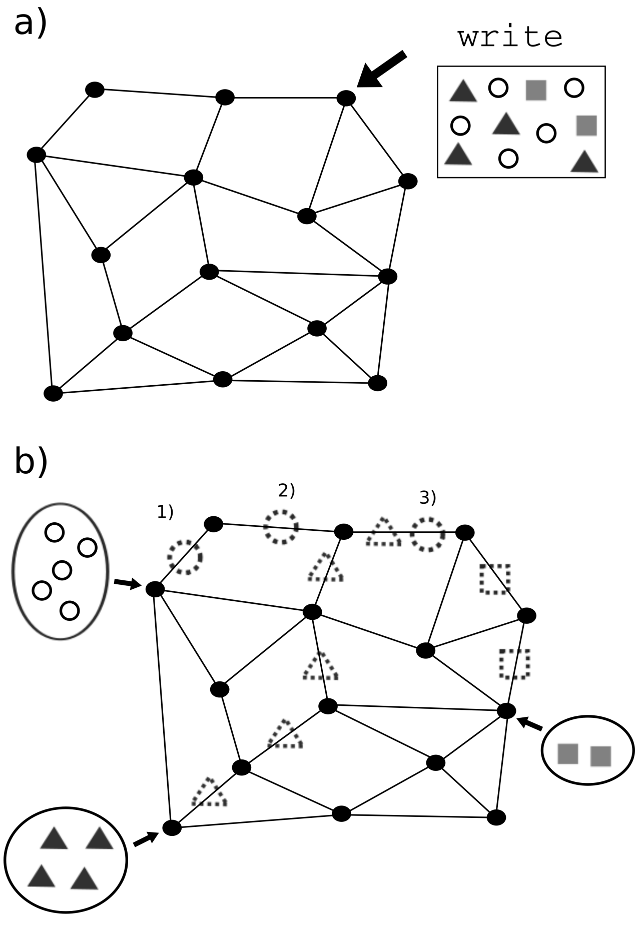

Figure 1: Visualization of the write algorithm

Figure 1 gives a simplified visualization of the write algorithm in which triples are illustrated as circles, squares and triangles. Triples which are considered to be similar by the applied similarity measure have the same shape. In a) a set of triples is written to one node in the network. As mentioned before each triple is stored three times in the network. For reasons of simplicity, in this example, we only consider the subject copy of the triple and omit the predicate and object copy. Thus for a triple (S,P,O) to be written an ant is created which explores the network in order to find a location where there are triples with subjects similar to S. After having stored its triple the ant takes its way back to the node where the write operation was performed leaving a pheromone trail in the system. b) shows a possible state in the system when the write procedure has finished. The dotted shapes on the edges depict the pheromone trails that the ants left on their ways back. For example, there are dotted circles marked with 1), 2), 3) from the node where all the circles have been stored to the node where the write operation was performed. Assuming that the circles are RDF triples that have a certain URI as subject the pheromones could be anything that stands for this URI, e.g. a hash value or some other numerical or string representation for the URI.

Algorithm 2 Read operation – basic algorithm

```
Require: Template $t$, hop count $h$
 1: while $h > 0$ do
 2:     $N = N \cup \{currentNodeId\}$
 3:     $T^t = findMatchingTriples(t)$
 4:     if not $empty(T^t)$ then
 5:         $spreadScentAndReturn(t, N)$
 6:         return $T^t$
 7:     else
 8:         $nextNode \leftarrow selectNextNode(T_i)$
 9:         $moveTo(nextNode)$
10:         $h = h - 1$
11:     end if
12: end while
13: $die()$
```

Read operation: Algorithm 2 describes the basic structure of the read operation. If a read request is issued to an arbitrary network node, a number of virtual ants is spawned to retrieve the triples requested by a triple template. These templates describe one component of a triple explicitly while leaving the other components open using wildcards, for example (ex:foo, ?p, ?s) for triples with "ex:foo" as an explicit subject. The ant is given a maximum hop count and starts searching for matching triples on the current node. If no triples are found, it selects and moves to a connected node using the scents present and a small random factor. If triples are found, the ant returns them to its origin while spreading scents similar to the write operation. Assuming that in figure 1 b) the circles stand for triples that have S as subject a read ant looking for triples matching the template (S, ?P, ?O) could make use of the dotted circle pheromones to find the S cluster.

Algorithm 3 Move operation – basic algorithm

```
Require: hop count $h$, drop limit $l_d$
 1: $T, i = pickupDissimilarTriple()$
 2: Remainder identical to algorithm 1 (write).
```

Move operation: Move operations are used to increase clustering within the network. Move ants are created if single nodes detect too much entropy in the data they store. The entropy is calculated using a similarity measure. Now some triples are moved to other nodes, triples "not fitting in" are more likely to be moved. Movement also can occur if a node is overloaded with too many triples. Apart from the way the triple is retrieved, the method used is identical to the write algorithm shown in Algorithm 3.

3.2 Benefits of swarm-based storage

The S4 concept offers a wide range of benefits compared to other approaches: Swarming individuals (ants) are based on simple algorithms, they do not require a complex ruleset to perform well, additionally, all decisions can be made using only a strictly local view. Still, individuals are able to dynamically adapt to their possibly changing environment. Network organization thus is decentralized and robust to changes in the network topology. In contrast to hash-based approaches such as DHT or B-Tree our concept does not require costly network reconstruction (achieved through communication) in the case of an error. Every node has sufficient information in the form of present scents to execute every possible query on its own, hence further eliminating single points of failure. Feasible solutions exist for issues like over-clustering, where a skewed data distribution leads unfair distribution of data storage [4]. The same is true for hot spot avoidance: if a node stores triples used in a large number of requests, it can simply move parts of them to another node or reject storage of similar triples in the future.

4. IMPLEMENTATION

While the described algorithms represent a very useful abstraction, an actual system implementing them has to consider "real-world" constraints such as limited memory, operation processing overhead, and network transmission limitations. We therefore designed several enhancements to the presented concept, which are described in the following sections.

4.1 Similarity calculation using clusters

The calculation of the similarity between the triple carried and the triples stored on a particular node would call for a comparison between all those triples. For this comparison to be efficient, we have extended the definition of our similarity measurement metric to yield an absolute value on a continuous scale for a given triple. "Similar" resources are assigned similar values on this scale. This enables our system to locally cluster similar values, a statistical mean for those clusters can also be calculated. The local similarity can now be calculated efficiently by using the local cluster means weighted by the amount of triples stored. This also improves efficiency on the move operation: instead of finding the most dissimilar triples the smallest local cluster can be moved to other parts of the network instead. Within main memory, a single local cluster only consist of four values continuously updated: cluster minimum value, cluster maximum value, cluster mean and element count. The actual triples can be written to persistent storage and are not being considered for similarity calculations.

The set of local clusters is required to be continuously updated as new statements are written and removed on the current node. Therefore, cluster algorithms that rely on the availability of all data for cluster generation cannot be used. We have constructed a different algorithm: For every triple to be stored, a lookup is performed within the cluster set with the corresponding cluster value calculated using our similarity function. If a cluster fits the current value (easily determined by comparing each clusters minimum and maximum value), the triple is added to the cluster. If no cluster is found, a new cluster is created containing the triple. In order to achieve clustering, a cluster merge operation is performed whenever the amount of clusters exceeds the maximum cluster amount specified. This process is described in algorithm 4: At first, the clusters containing only a small amount of elements are merged, this is controlled by the s_{max} size limit first initialized with the minimum value. Then, the two clusters with less elements than s_{max} and the smallest distance between their means are identified. These are then merged. If there are no two clusters matching this condition, s_{max} is doubled and the merge process is restarted. The whole algorithm runs until the amount of clusters falls below the maximum cluster amount m.

To illustrate this approach, figures 2 and 3 show the local clusters formed using a string-distance based similarity measure and our clustering algorithm as described. Each bar represents a local cluster. The bar width is determined by the cluster minimum and maximum values and its height is given by the amount of triples within that cluster. Both cluster sets were limited to 20 clusters, however, clusters containing less than 100 entries do not show at this scale. Figure 2 illustrates clusters formed from storing each component of 100K statements taken from the DBpedia dataset [10] which is extracted from Wikipedia. Figure 3 shows the clusters formed from the same amount of statements, but this time generated by LUBM's test data generator, a synthetic benchmark [8].

Algorithm 4 Cluster merging

```
Require: a set of clusters $S$
Require: cluster set size limit $m$
 1: $s_{max} \leftarrow 1$
 2: repeat
 3:   $d_{min} \leftarrow 1$
 4:   for each distinct pair of clusters $(a, b)$ do
 5:     $d_{a,b} \leftarrow distance(a, b)$
 6:     if $(d_{a,b} < d_{min}) \wedge (|a| <= s_{max} \vee |b| <= s_{max})$ then
 7:       $d_{min} \leftarrow d_{a,b}$
 8:       $mp_a \leftarrow a$
 9:       $mp_b \leftarrow b$
10:     end if
11:   end for
12:   if $d_{min} < 1$ then
13:     $c_{a,b} \leftarrow mp_a \cup mp_b$
14:     $S \leftarrow (S \setminus \{mp_a, mp_b\}) \cup c_{a,b}$
15:   end if
16:   $s_{max} \leftarrow s_{max} * 2$
17: until $|S| < m$
```

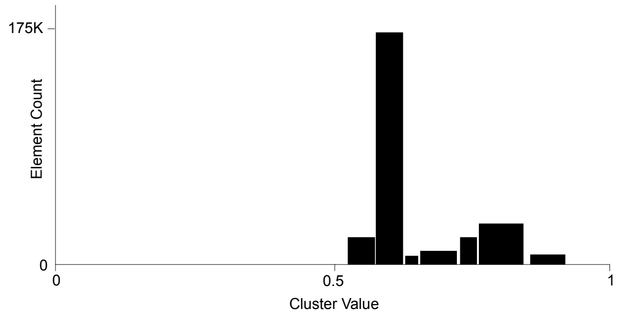

Figure 2: Local clusters for DBpedia dataset

Two observations can be made from these figures: First, local clustering is indeed performed by our algorithm, and second, cluster distribution and width entirely depends on the data to be stored. The results were expected, LUBM's synthetic benchmark data only contain a very small distribution of values compared to DBpedia.

We have extended local clustering to the in-memory storage of the virtual pheromones used by our virtual ants to optimize their retrieval and write strategies. These volatile pheromone values are now also stored in a clustered manner, which is made possible by the absolute mapping provided by the similarity metric. Pheromone values decay as they age. Instead of periodically updating the pheromone table, we have chosen to store the time of the last update with the pheromone, enabling on-demand calculation of the current value and eliminating the need for costly updates.

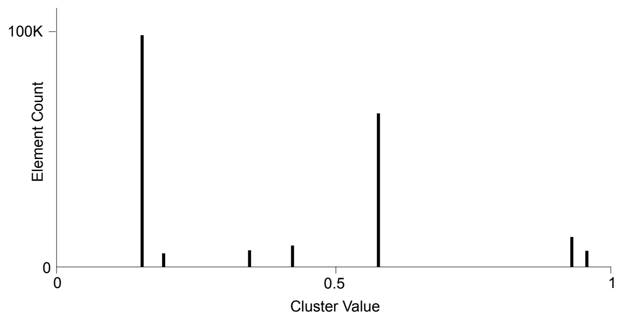

Figure 3: Local clusters for LUBM dataset

4.2 Transmission optimizations

All triples to be written are re-organized into groups sharing the same clustering keys. That way, one ant performing a write operation is able to carry multiple triples to their destination. This greatly reduces system load, as computational costs for ant operations over multiple nodes greatly outweigh the additional memory requirement on the node handling the client request. However, this also increases the amount of data transmitted over the network for a single virtual ant. While the loss of a single ant during transmission may be tolerated, the loss of an ant carrying 10000 statements cannot be. As a consequence, ant movement from one node to another is monitored carefully, and rolled back if necessary. This means that if a transmission between two nodes fails the sender restarts the whole transmission. The receiver only processes fully transmitted messages.

When triples are read from the store, the read-ants are likely to encounter potentially large sets of triples on a single node matching the read pattern. To increase read performance in these cases, not only is a single ant able to handle all locally found triples, this set is also directly transmitted to the node the read request originated from. The ant then only tracks back its path in order to increase the scents on the way, but with its operations heavily accelerated without the burden of the read result set.

4.3 System Architecture

Figure 4 shows our system architecture: Requests are taken from the client by the *Client API*, the different operations are differentiated and then forwarded to the corresponding methods within the *Query Processor*. Our virtual ants are modeled as events within the system, thus the Query Processor converts the operations into events, which are then processed by the *Event Handlers*. These components contain our different algorithms, including the already mentioned algorithms for read, write and move. Operations modifying stored triples do so within the *Tuple Storage* component, which is backed by an on-disk triple store. If an event is to be transmitted to another node, the *Routing* component selects a node from the neighbor list and transmits the event to this node. There is no conceptional difference between events received from other nodes and events created locally by the query processor, this keeps the algorithms concise.

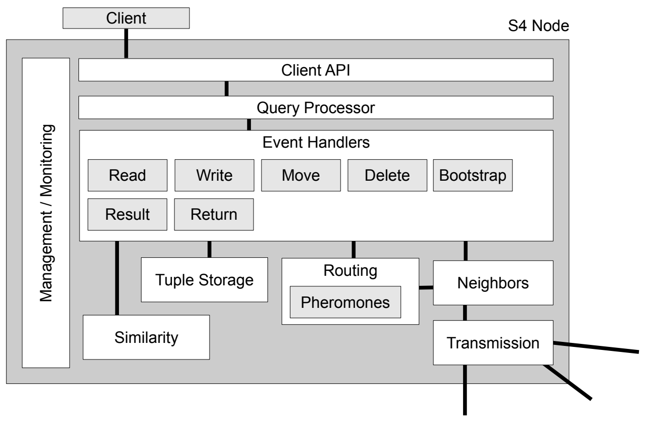

Figure 4: S4 System architecture

5. EVALUATION

We have implemented the aforementioned concepts of the S4 system and its practical enhancements such as clustered similarity values using the Java programming language as a full stand-alone system. This implementation then was deployed onto our test cluster consisting of 150 virtual Linux nodes running on a single server with eight 2.26 GHz processors and 64 GB of main memory. Each node was given a number of neighbors using a simple bootstrap protocol. For a typical run, a node had on average 11 neighbor nodes. The following evaluation is focused on the amount of hops a single read operation needs to perform to locate an arbitrary triple stored within the system. Storage networks can be comprised of many different amounts of nodes, thus we repeated the evaluation for a set of nodes ranging from 20 to 150 nodes in total. For the purpose of evaluation, the read operation was extended to carry a detailed trace of its journey through the network. A random subset of the DBpedia dataset containing 100K triples was stored inside the storage network for each test run. Each test run consisted of the following steps:

1. Start up the current amount of nodes
2. Write the dataset to a random node, measure the time
3. Perform the read operation on each running node, measure the amount of hops
4. Collect system status from every node
5. Stop nodes and delete all stored data

Alhough the dataset was written to a single node, the data is stored distributed over all nodes of the storage network, due to the swarm-based clustering. Figure 5 shows the amount of triples stored on each node for a storage network of 150 running nodes. The spikes denote large clusters of very similar resources, for example of the `rtf:type` property, which is used in roughly 10% of the triples in our test data set.

5.1 Write operation performance

For each network configuration, our DBpedia subset containing 100K triples was written to a single arbitrary node. The time required to complete the write operation from the client standpoint was recorded for each configuration. Table 1 contains the results. A tendency to the expected system behaviour can for example be

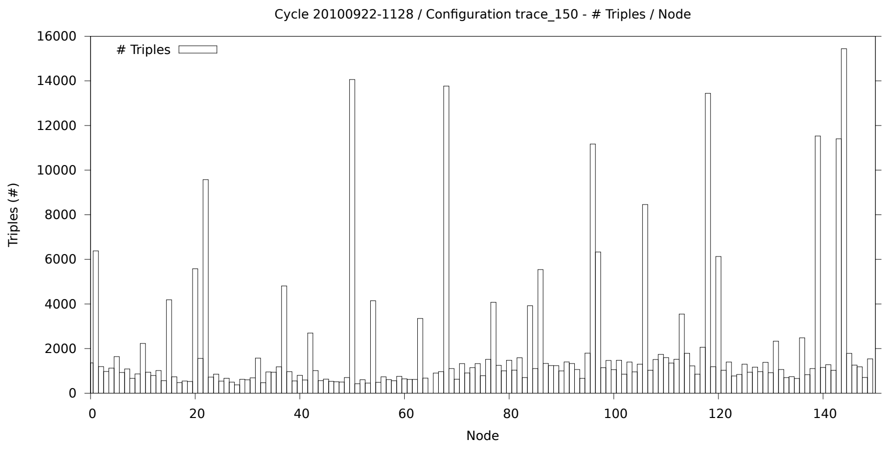

Figure 5: Triples per node for a storage network of 150 nodes

observed by comparing the results for 40 and 80 nodes, respectively. Even though the amount of nodes (and thus storage capacity) have doubled, the amount of time required to write the test data set into the network has increased only by a comparably small amount. This increase is attributed to the larger search space for the data clustering inside the network, should more nodes be used.

nodes (#)	write time (s)
20	503
30	478
40	500
50	529
60	589
70	555
80	640
90	666
100	678
110	703
120	697
130	733
140	748
150	769

Table 1: Write time for different network sizes

5.2 Read operation performance

The amount of hops required to retrieve a particular item is the most prevalent and independent metric to evaluate read performance in a distributed system. Thus, for every network size tested, we issued a read operation on all nodes. The read operations were monitored for the amount of hops taken before a match was found. From this set of hop counts per node for a single network configuration the average and median statistical indicators were calculated. If pheromone paths are not developed yet, a read operation could also fail. Thus, the percentage of nodes which have successfully retrieved a result has also been recorded. Figure 6 shows typical results of the evaluation for the read operation. The different network sizes are given on the x-axis. For each network size, average (+ symbol) and median (x symbol) measures of the amounts of hops from all nodes in the current network are plotted on the left y-axis. The percentage of nodes that have successfully returned a result is plotted ($\star$ symbol) on the right y-axis.

Due to the randomness inherent in our swarm-based approach, this evaluation results fluctuate to some degree. However, the result graph pictured is typical for the system performance, and a general trend is observable, where the double amount of nodes used does neither lead to the same increase in hop count, nor to a significantly decreased response rate. Further evaluations would then extend our concept to a larger number of hosts, and a wider variety of data sets and data set sizes.

The execution of a read operation requires a set of routing decisions, which are now detailed further. The following list shows an actual read operation for a key `key1` in a storage network consisting of 150 nodes. A read operation is started on the node with the id 13. Since no results are found on the local node, the present pheromones are used to determine the next node to visit. The pheromone intensity is dependent on the frequency and amount of triples found. For this key, and the closer an operation moves to the node actually storing the triple, the stronger the pheromone intensity are getting. This effect is due to the concentration of read operations for this particular key in the vicinity of the actual storage location. Following the pheromone trail, the operation visits seven nodes before successfully locating results on node 50.

After results have been found, they are forwarded to the node the query originated from according to our implementation optimization. Nevertheless, the path taken is tracked back to increase the pheromone intensity for `key1`. Intensity is increased on the network connections towards the node storing the corresponding triples. Relevant parts of the operation trace are listed, each entry starts with the node id the operation is currently executed on.

- 13 - Read operation for `key1`, no local results, node 69 as next hop with intensity 0.55

...

- 118 - No local results, node 50 as next hop with intensity 11.97
- 50 - Found 4 results for `key1`, returning results to node 13, track back spreading pheromones for `key1`, node 118 as next hop
- 118 - Intensify to node 50, node 62 as next hop

...

- 13 - Intensify `key1` to node 69, terminate

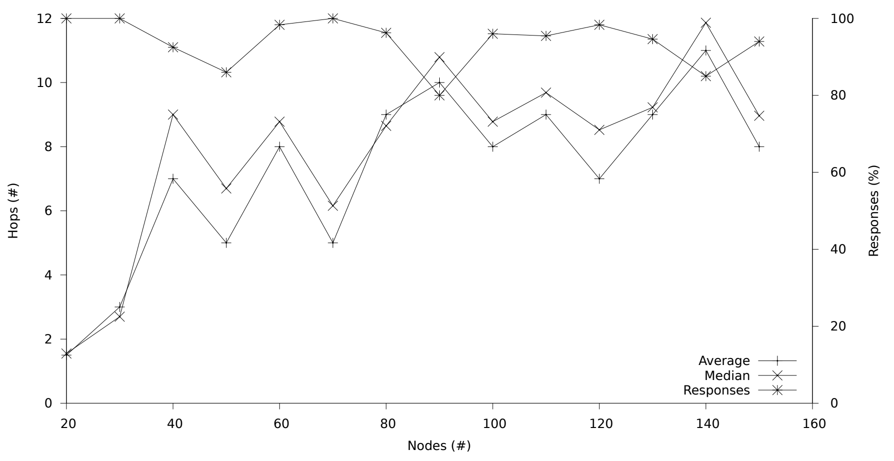

Figure 6: Hops required to read a triple for different network sizes

6. CONCLUSION AND OUTLOOK

We have described our motivation to extend the current state of storage systems for Semantic Web applications. The differences between centralized and distributed semantic storage services have been shown, as well as the current research in storage systems. We have described our design of a Self-Organized Semantic Storage Service (S4), its basic algorithms and advantages over conventional data distribution algorithms. The current status of our S4 implementation was outlined, especially our optimizations of the basic algorithms for real-world application. Our preliminary evaluation of our implementation using a physical cluster of nodes showed the general validity of our approach, as well as its potential benefits.

While reasoning on central storage systems can be performed using logic engines, distributed reasoning is far more complex. As a general approach, achieving reasoning completeness is sacrificed in favor of scalability for distributed reasoning. Current development occurs within the Large Knowledge Collider (LarKC) Project [6], which aims to enable reasoning for datasets containing up to a trillion triples. Consequently, the next level for S4 is to design and implement a concept for a simple reasoning during retrieval along an is-a hierarchy and with a best-effort service contract. We then plan to add application-specific reasoning capabilities. Our use-case for that is a system utilizing geo-information and we will use semantic geometric relations.

We will continue working on the refinement and implementation of our S4 concept within our current joint research project "DigiPolis", where this system will form the basis for an indoor navigation system possibly covering entire cities. Example use cases include a semantic search for points of interest in a vicinity, in-house routing with semantic restrictions, and semantic annotation of map entries.

Acknowledgment

This work has been partially supported by the "DigiPolis" project funded by the German Federal Ministry of Education and Research (BMBF) under the support code 03WKP07B.

7. REFERENCES

[1] Eric Bonabeau, Marco Dorigo, and Guy Theraulaz. *Swarm Intelligence: From Natural to Artificial Systems*. Santa Fe Institute Studies in the Sciences of Complexity Series. Oxford Press, July 1999.

[2] Min Cai and Martin Frank. RDFPeers: a scalable distributed RDF repository based on a structured peer-to-peer network. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 650–657, New York, NY, USA, 2004. ACM.

[3] Jeremy J. Carroll, Ian Dickinson, Chris Dollin, Dave Reynolds, Andy Seaborne, and Kevin Wilkinson. Jena: Implementing the semantic web recommendations. Technical Report HPL-2003-146, Hewlett Packard Laboratories, December 24 2003.

[4] Matteo Casadei, Ronaldo Menezes, Robert Tolksdorf, and Mirko Viroli. On the problem of over-clustering in tuple-based coordination systems. In *SASO*, pages 303–306. IEEE Computer Society, 2007.

[5] Philippe Cudré-Mauroux, Suchit Agarwal, and Karl Aberer. GridVine: An infrastructure for peer information management. *IEEE Internet Computing*, 11(5):36–44, 2007.

[6] Dieter Fensel, Frank van Harmelen, and Bo Andersson. Towards LarKC: A platform for web-scale reasoning. In *ICSC*, pages 524–529. IEEE Computer Society, 2008.

[7] David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7:80–112, 1985.

[8] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: A benchmark for OWL knowledge base systems. *J. Web Sem*, 3(2-3):158–182, 2005.

[9] Andreas Harth, Jürgen Umbrich, Aidan Hogan, and Stefan Decker. YARS2: A federated repository for querying graph structured data from the web. In *The Semantic Web, ISWC 2007, Busan, Korea, November 11-15, 2007*, volume 4825 of *Lecture Notes in Computer Science*, pages 211–224. Springer, 2007.

[10] Ted Thibodeau Jr. The DBpedia data set. Online *http://wiki.dbpedia.org/Datasets*, November 2009.

[11] Atanas Kiryakov, Damyan Ognyanov, and Dimitar Manov. OWLIM - A pragmatic semantic repository for OWL. In *WISE 2005 Workshops, New York, NY, USA, November*

20-22, 2005, Proceedings, volume 3807 of *Lecture Notes in Computer Science*, pages 182–192. Springer, 2005.

[12] Frank Manola and Eric Miller, editors. *RDF Primer*. W3C Recommendation. World Wide Web Consortium, February 2004.

[13] Ronaldo Menezes and Robert Tolksdorf. A new approach to scalable linda-systems based on swarms. In *Proceedings of ACM SAC 2003*, pages 375–379, 2003.

[14] Wolfgang Nejdl, Boris Wolf, Changtao Qu, Stefan Decker, Michael Sintek, Ambjrn Naeve, Mikael Nilsson, Matthias Palmr, and Tore Risch. EDUTELLA: A P2P networking infrastructure based on RDF. *Proceedings of the eleventh international World Wide Web Conference*, January 01 2002.

[15] Ontotext AD. OWLIM benchmarking: LUBM. Online *http://www.ontotext.com/owlim/benchmarking/lubm.html*, February 2010.

[16] Robert Tolksdorf and Anne Augustin. Selforganisation in a storage for semantic information. *Journal of Software*, 4, 2009.

[17] Robert Tolksdorf, Anne Augustin, and Sebastian Koske. Selforganization in distributed semantic repositories. *Future Internet Symposium 2009 (FIS2009)*, 2009.

[18] Jing Zhou, Wendy Hall, and David De Roure. Building a distributed infrastructure for scalable triple stores. *J. Comput. Sci. Technol*, 24(3):447–462, 2009.