SOMO: Self-Organized Metadata Overlay for Resource Management in P2P DHT

Zheng Zhang♦ Shu-Ming Shi♣ and Jing Zhu♣

Abstract – *In this paper, we first describe the concept of data overlay, which is a way to implement arbitrary data structure in a structured P2P DHT. Built on top of that, we developed a self-organized and robust infrastructure, called SOMO, to perform resource management in an arbitrary DHT. It does so by gathering and disseminating system metadata in* O(log*N*) *time with a self-managed and self-survivable data overlay. Our preliminary results of using SOMO to balance routing traffic with node capacities in a prefix-based overlay have demonstrated the potential of both these two techniques.*

1 Introduction

For a large scale P2P overlay to adapt and evolve, there must be a parallel infrastructure to monitor the health of the system (e.g, "top"-like utility in UNIX). The responsibility of such infrastructure is to gather from and distribute to entities comprising the system whatever system metadata of concern, and possibly serve as the channel to communicate various scheduling instructions. The challenge here is that this infrastructure must be embedded in the hosting DHT but is otherwise agonistic to its specific protocols; it must grow along with the hosting DHT system; it must also be fault resilient and, finally, the information it gather and/or disseminate should be as accurate as possible.

In this paper, we describe the *Self-Organized Metadata Overlay*, or SOMO in short, which accomplishes the above goal. By using hierarchy as well as soft-state, SOMO is fault-resilient and can gather and disseminate information in O(log*N*) time. SOMO is simple and flexible, and is agnostic to both the hosting P2P DHT and the data being gathered and disseminated. The later attribute allows it to be programmable, invoking appropriate actions such as merge-sort and aggregation as data flows through.

Through the development of SOMO, we have discovered that there is a consistent and simple mechanism to implement arbitrary data structure on top of a P2P DHT. We refer to a data structure that is distributed onto a DHT a *data overlay*. Data overlay is discussed in Section-2. Following that, we describe the construction and operations of SOMO in Section-3, and also its application in Section-4. A case study of using SOMO to balance routing traffic to node's capacity in a prefix-based overlay is offered in Section-5, along with preliminary results. We discuss related work in Section-6 and conclude in Section-7.

♦ Microsoft Research Asia. zzhang@microsoft.com

♣ Tsinghua university. Work done as intern at MSRA {ssm99, zhujing00}@mails.tsinghua.edu.cn

2 Implement arbitrary data structures on top of P2P DHT using data overlay

We observe that hash-table is only one of the fundamental data structures. Sorted list, binary trees, queues all have their significant utilities. One way would be to investigate how to make each of them self-organized (i.e., P2P sorted list). Another is to build on top of a hash table that is already self-organizing (i.e. P2P DHT). This second approach is what we take in this paper.

Any object of a data structure can be considered as a document. Therefore, as long as it has a key, that object can be deposited into and retrieved from a P2P DHT. Objects relate to each other via pointers, so to traverse to the object *b* pointed by *a.foo*, *a.foo* must now store *b*'s key instead. More formally, the following two are the necessary and sufficient conditions:

- Each object must have a key, obtained at its birth (i.e., *new*)
- If an attribute of an object, *a.foo*, is a pointer, it is expanded into a structure of two fields: *a.foo.key* and *a.foo.host*. The second field is a soft state containing the last known hosting DHT node of the object *a.foo* points to. It is thus a routing shortcut.

It is possible to control the generation of object's key to explore data locality in a DHT. For instance, if the keys of *a* and *b* are close enough, it's likely that they will be hosted on one machine in DHT.

We call a data structure distributed in a hosting DHT a *data overlay*. It differs from traditional sense of overlay in that traversing (or routing) from one entity to another uses the free service of the underlying P2P DHT.

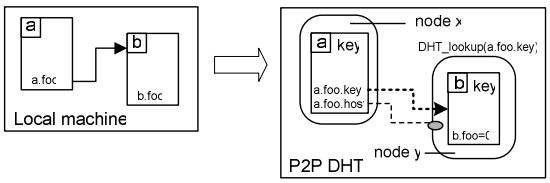


Figure 1: implement arbitrary data structure in DHT

Figure-1 contrasts a data structure in local machine versus that on a P2P DHT. Important primitives that manipulate a pointer in a data structure, including *setref*, *deref* (dereferencing) and *delete*, are outlined in Figure-2.

Here, we assume that both DHT_lookup and DHT_insert will, as a side effect, always return the node in DHT that currently hosts the target object. DHT_direct bypasses normal DHT_lookup routing and directly seeks to the node that hosting an object given its key.

```
setref(a.foo, b) {      // initially a.foo=null
                        // b is the object to which a.foo
                        // will point to
  a.foo.key = b.key
  a.foo.host = DHT_insert(b.key, b)
}
deref(a.foo) {          // return the object
                        // pointed by a.foo
  if (a.foo ≠ null) {
    obj = DHT_direct(a.foo.host, a.foo.key)
    if obj is null  // the object has moved
      obj = DHT_lookup(a.foo.key)
      a.foo.host = node returned
    }
    return obj
  } else return "non-existed"
}
delete(a.foo) {    // delete the object pointed by a.foo
  DHT_delete(a.foo.key)
  a.foo=null
}
```

Figure 2: pointer manipulate primitives

The interesting aspect is that it is now possible to host any arbitrary data structure on a P2P DHT, and in a transparent way. What need to be modified are the library routine that creates an object to insert a key as its member, and the few primitives that manipulate pointers as outlined. Therefore, legacy applications can be ported to run on top of a P2P DHT, giving them the illusion of an infinite storage space (here storage can broadly include memory heaps of machines comprising the DHT).

3 Self-Organized Metadata Overlay

We now describe the data overlay SOMO (*Self-Organized Metadata Overlay*), an information gathering and disseminating infrastructure on top of any P2P DHT. To recap briefly, such an infrastructure must satisfy a few key properties: *self-organized* at the same scale as the hosting DHT, fully *distributed* and *failure-resilient*, and be as *accurate* as possible of the metadata gathered.

SOMO is a tree of k degree and its leaves are planted in each DHT node. Information is gathered from the bottom and propagates towards the root, and disseminated by trickling downwards. Thus, one can think of SOMO as doing *converge cast* from the leaves to the root, and then *multicast* back down to the leaves again. Both the gathering and dissemination phases are $O(\log_k N)$ bounded, where N is total number of entities. Each operation in SOMO involves no more than $k+1$ interactions, making it fully distributed. We deal with robustness using the principle of soft-state, so that data can be regenerated in $O(\log_k N)$ time. The SOMO tree is both self-organized and self-survivable and can automatically reconstruct itself in the same time bound. We explain the details of SOMO in the following sub-sections.

3.1 Building SOMO

Since SOMO is a tree, we call its node the *SOMO node*. To avoid confusion, we denote the DHT nodes as simply the *DHT node*. A DHT node that hosts a SOMO node s, is referred to as *DHT_host*(s).

```
struct SOMO_node {
  string key
  struct SOMO_node *child[1..k]
  DHT_zone_type Z
  SOMO_op  op
  Report_type report
}
```

Figure 3: SOMO node structure

The basic structure of the type *SOMO_node* is described in Figure-3. The member Z indicates the region of which this node's *report* member covers. Here, the region is simply a portion of the total logical space of the DHT. The root SOMO node covers the entire logical space. The *key* is the center of a SOMO node's region. Therefore, a SOMO node s will be hosted by a DHT node that covers $s.key$ (i.e. the center of $s.Z$). A SOMO node's responsible region is further divided by a factor of k, each taken by one of its k children, which are pointers in the SOMO data structure. A SOMO node s's i-th child will cover the i-th fraction of region $s.Z$. Since a DHT node will own a piece of the logical space, it is therefore guaranteed a SOMO node will be planed in it.

Initially, when the system contains only one DHT node, there is only the root SOMO node. As the DHT system grows, SOMO builds its hierarchy along. This is done by letting each SOMO node periodically execute the routine *SOMO_grow*, shown in Figure-4. .

```
SOMO_grow (SOMO_node s) {
        // check if any children is necessary
  if (s.Z ⊆ DHT_host(s).Z)  return
  for i=1 to k
    if (s.child[i] is null &&
        the i-th sub-space of s.Z ⊄ host(s).Z ) {
      t = new(type SOMO_node)
      t.Z = the i-th sub-space of s.Z
      t.key = center of t.Z
      setref(s.child[i], t)    // inject into DHT
    }
}
```

Figure 4: SOMO_grow procedure

We test first if the SOMO node's responsible zone is smaller or equal to that of the hosting DHT node's, if the test comes out to be true, then this SOMO node is already a leaf planted in the right DHT node and there is no point to grow any more child. Otherwise, we attempt to grow. Note that we initialize a SOMO node object and its appropriate fields, and then call the *setref* primitive (See

Figure-2) to install the pointer; this last step is where DHT operation is involved.

As this procedure is executed by all SOMO nodes, the SOMO tree will grow as the hosting DHT grows, and the SOMO tree is taller in logical space regions where DHT nodes are denser. This is illustrated in Figure-5.

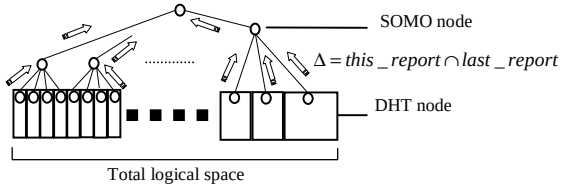


Figure 5: SOMO tree on top of P2P DHT

The procedure is done in a top down fashion, and is executed periodically. A bottom-up version can be similarly derived. When system shrinks, SOMO tree will prune itself accordingly. This can be done by simply deleting redundant children.

The crash of a DHT node will take away the SOMO nodes it is hosting. However, the periodical checking of all children SOMO nodes ensures that the tree can be completely reconstructed in $O(\log_k N)$ time. Because the SOMO root is always hosted by the DHT node that owns the center of the total space, we ensure the existence of the SOMO root by letting that DHT node invoke the SOMO_grow routine on the SOMO root.

3.2 Gathering and disseminate information with SOMO

To gather system metadata, for instance loads and capacities, a SOMO node periodically requests report from its children. The leaf SOMO nodes simply get the required info from their hosting DHT nodes. As a side-effect, it will also re-start a child SOMO node if it has disappeared because the hosting DHT node's crash. The following figure illustrates the procedure.

```
get_report (SOMO_node s) {
  Report_type rep[1..k]
  for i∈[1..k]
    if (s.child[i] ≠ null)          // retrieving via DHT
      rep[i] = deref(s.child[i]).report
  s.report = s.op(rep[])
}
```

Figure 6: SOMO gathering procedure

The routine is periodically executed at an interval of T. Thus, information is gathered from the SOMO leaves and flows to its root with a maximum delay of $\log_k N \cdot T$. This bound is derived when flow between hierarchies of SOMO is completely unsynchronized. If upper SOMO nodes' call for reports immediately triggers the similar actions of their children, then the latency can be reduced to $T+t_{hop} \cdot \log_k N$, where t_{hop} is average latency of a trip in the hosting DHT. The unsynchronized flow has latency bound of $\log_k N \cdot T$, whereas the synchronized version will be bounded by T in practice (e.g., 5 minutes). Note that $O(t_{hop} \cdot \log_k N)$ is the absolute lower bound. For 2M nodes and with k=8 and a typical latency of 200ms per DHT hop, the SOMO root will have a global view with a lag of 1.6s.

Dissemination using SOMO is essentially the reverse: data trickles down through the SOMO hierarchy towards the leaves. Performance thus is similar as gathering. By some modification, dissemination can piggyback on the return message in the gathering phase.

Operations in either gathering or disseminating phases involve one interaction with the parent, and then with k children. Thus, the overhead in a SOMO operation is a constant. The entities involved are the DHT nodes that host the SOMO tree. SOMO nodes are scattered among DHT nodes and therefore SOMO processing is distributed and scales with the system.

It seems that towards the SOMO root the hosting DHT nodes need to have increasingly higher bandwidth and stability. As discussed earlier, stability is not a concern because the whole SOMO hierarchy can be recovered in $O(\log_k N)$ time. As for bandwidth, most of the time one needs only to submit deltas between reports, and this will bring down message size significantly (Figure-5). Finally, it is always possible to locate an appropriate node through SOMO. This node can swap with the one who is hosting the SOMO root currently. That is to say, SOMO can be completely self-optimizing as well.

3.3 Discussion

The SOMO procedures are just like any ordinary code running on a local machine. This demonstrates the utility of the data overlay concept.

The power of SOMO lies in its simplicity and flexibility. As an infrastructure, SOMO does not specify what information it should gather and/or disseminate. Furthermore, various operations, such as aggregation and sorting, can be invoked when metadata flows by. That is to say, SOMO operations are programmable and *active*. For this reason, in the pseudo-code we use *op* as a generic notation for operation used. Some of the examples will be described in the next Section.

4 Application of SOMO

As a scalable, fault-tolerant metadata gathering and dissemination infrastructure, the utilities of SOMO are many. In a large scale system, the need to monitor the health of the system itself can not be understated. More advanced usages are chiefly decided by algorithms that built upon the metadata that gathered. The followings are some examples.

One instance would be to find powerful nodes, commonly known as *supernodes*. To do this, we will make a SOMO tree where the report type is sorted list, and the *op* is merge-sort. Thus, SOMO can mine out multiple classes of supernodes, as reports available at various internal SOMO nodes, with the SOMO root having the complete list. These supernodes can thus act as indexing hobs, as proposed in [2]. Supernodes can

form separate overlays to optimize routing. Given a node's own geographic coordinates, we may wish to find out the closest node in a given logical space region. This is particularly useful for the *proximity neighbor selection* optimization in prefix-based overlay such as Pastry[5], Tapestry[12] and eCAN[10]. To do this, the leaf SOMO node submits its coordinates (obtained perhaps via a method like land-marking [4]) in the SOMO report, and the SOMO *op* is aggregation, and any node geographical distribution in a region can be found by query the appropriate SOMO node that covers that region.

More elaborate algorithms can be built upon load and capacity info fed by a SOMO to perform various kind of load balancing. One such application is offered in the next section. There are proposals where routing performance is the best but storage uniformity is sacrificed [4], SOMO can discover the density map of node capacities. Such information can guide document placement, or migrate nodes from dense regions to weak ones. In this way, uniformity will improve over time.

It is also possible to build a SOMO on top of a basic, mesh-based P2P DHT, and then build a O(log*N*) soft-state prefix-based overlay because SOMO can have the knowledge of what nodes exist in what portion of the total logic space.

Finally, since SOMO is a self-organized hierarchy, by installing appropriate filters as its *op* member, it's possible to use SOMO as a pub/sub infrastructure as well.

5 Case study: balancing routing power with routing traffic in prefix-based overlay

Prefix-based overlay includes Tapestry[12], Pastry[5], Chord[7], Kademlia[1] and eCAN[9] (CAN[3] with simple extension). Though some aspects of these proposals differ, they share a few key attributes: 1) the total logical (or key) space is recursively divided and 2) routing greedily seeks out the biggest span into a sub-space and then zoom in towards target quickly. Routing table of prefix-based overlay is an array, recording spaces of exponentially decreasing size and one or several nodes that serve as this node's gateway, or "router" into these spaces. The flexibility of the prefix-based overlays is that, any node in the target sub-space can be a router. This gives rise to many optimization opportunities. Pastry and eCAN explore the possibilities of using the geographically closest node as router candidates to improve routing performance. In this paper, we report our investigation on another complementary axis: choosing the more powerful nodes to serve as routing entrances for larger sub-spaces where traffics are exponentially more than sub-spaces enclosed. The ultimate solution (and challenge) of selecting these "routers" is to consider all the following three factors: geographic vicinity, routing capacity and load distribution. This remains to be one of our future works.

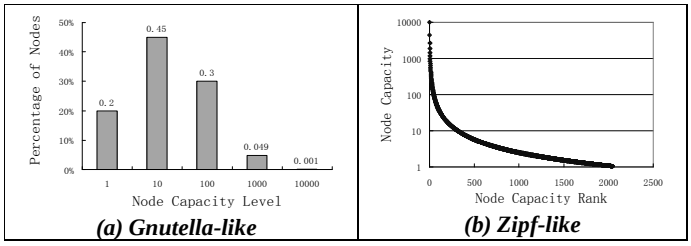
Our goal is to promote the most capable nodes to handle traffics into larger space, relieving weaker nodes off these responsibilities. Intuitively, the most powerful nodes will take the bulk of the loads in the largest enclosing space, and the weaker ones will serve no more than those designated to its immediate neighbor. Due to space limitation, we refer readers to [10] for the full protocol. Our basic idea is to classify routing loads that a node takes according to the sub-space in which the routing is designated and divide a node's routing capacity accordingly.

Our optimization consists of four algorithms:

- ***Statistic collection algorithm***. Aggregate loads and capacity statistics in a bottom-up sweep through SOMO. The goal is to have a "view" of the demographic distribution of both loads and capacities. At this point, the load/capacity ratios of the whole system as well as all enclosing spaces are available.
- ***Load balance algorithm***. Top-down sweep to determine the amount of routing capacities to be dedicated in each space, so that its load/capacity ratio approaches to that of the whole system where possible.
- ***Capacity selection algorithm***. Select the right portion of capacities, as recommended by the previous step, from candidate nodes. Also bottom-up sweep. At the end of this algorithm, we have selected the right capacity divide to take care of traffic loads of different space.
- ***Entries dissemination algorithm***. Notify other nodes to use these new "routers" so that load distribution can take effect.

The core of our algorithm is in the 2nd and the 3rd step. It can be simplified if not for a subtle but important issue. The load and capacity distribution can be so skewed that nodes in a sub-space are already overwhelmed by the traffic designated to them, leaving them virtually no surplus power to share routing duties in enclosing spaces. Our scheduling algorithm has taken this into full account by pardoning heavily loaded sub-spaces (or the ones with meager power).

We modify an earlier e/CAN simulator by incorporating all the four algorithms described earlier. eCAN[9] is a prefix-based overlay capable of O(ln*N*) routing performance, and this is achieved with simple extension to CAN[2]. In eCAN, the recursion in resolving routing is by zooming into topology sub-zones rather than shifting bits. However, our algorithm is immediately applicable to other prefix-based overlays such as Pastry[5], Tapestry[12] and Kademlia[1]



(a) Gnutella-like *(b) Zipf-like*

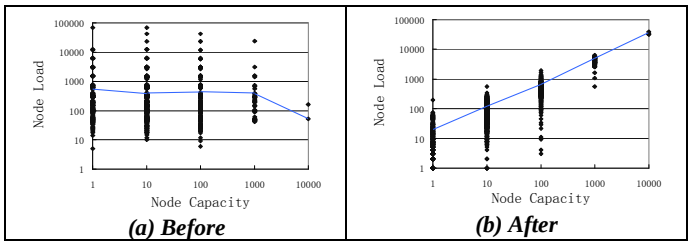
Figure 7: Capacity profile (*N*=2K)

Two capacity profiles are used to model heterogeneity:

- **_Zipf-like_**: when sorted, the i-th node has capacity $10000 \cdot i^{-\beta}$, we choose β be 1.2 by default.
- ***Gnutella-like:*** there are 5 levels of node, and the i-th level has capacity 10^{i-1}, popularity in these levels are 20%, 45%, 30%, 4.9% and 0.1%, going from level 1 to level 5 (see [6]).

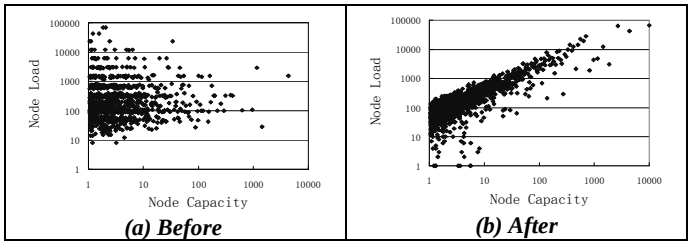
The comparison of the two distributions for a 2K node system is shown in Figure 7.

The eCAN configuration we use is equivalent to Pastry/Tapestry of *b*=1. We tested other configurations [10] and results are similar to those presented here. For each configuration (capacity profile, *N* and other parameters), an experiment of 5 cycles is run. Each cycle starts with a complete reshuffling of the node capacities, then route 100*N* times, during which load and capacity information are gathered. We then run the four algorithms to perform load balance. Finally another 100*N* routings are performed and various statistics are collected again. This somewhat primitive setup allows us to gain sufficient insight of the algorithms; a more sophisticated one would include node join and leave events and mix SOMO traffics with normal routing, which we plan to conduct in the future.



(a) Before *(b) After*

Figure 8: Results of *N*=2K, Gnutella-like (the line corresponds to average load of a capacity level)



(a) Before *(b) After*

Figure 9: Results of N=2K, Zipf-like

We found that, in all configurations, load balance converges quickly in *O*(log *N*) time, and that after the full set of the algorithms are run, higher capacity nodes are taking more loads. Figure 8 and Figure 9 show a typical pair of results of the Gnutella-like and Zipf-like capacity distributions, respectively. Note the sharp difference before and after the load redistribution.

6 Related work

Data overlay relies on the key property of the P2P DHT ([5] [12] [7] [3] [1] [9]) that an item with unique key can be reliably created and retrieved. To our knowledge, extending the principle of self-organizing to arbitrary data structure other than hash table and do it in a way that is agnostic to both semantics and performance of the hosting P2P DHT is new.

SOMO bears the most similarity to Astrolabe [8], a peer-to-peer management and data mining system, for instance the use of hierarchies and aggregation. SOMO operates at the rudimentary *data structure* level while Astrolabe is on a virtual, hierarchical *database*. SOMO's extensibility is much like that of active network, whereas Astrolabe uses SQL queries. The marked difference is that SOMO is designed specifically on top of P2P DHT, for two reasons: 1) we believe P2P DHTs have established a foundation over which many other systems can be built and thus there is a need for a scalable resource management and monitoring infrastructure and 2) by leveraging P2P DHT (in fact, data overlay) the design and protocols of such infrastructure can be much simpler.

7 Conclusion and future work

This paper makes several novel contributions: we describe how arbitrary data structures can be implemented on P2P DHT using the concept of data overlay; we designed and evaluated a self-organizing and robust metadata gathering and dissemination infrastructure, the *self-organized metadata overlay*. We have demonstrated how to balance routing traffic with node capacity in prefix-based overlay using both of these two techniques. Our future work includes more extensive study of these concepts.

8 Acknowledgement

The authors would like to thank Yu Chen and Qiao Lian for their useful comments of this work. Dan Zhao helped to prepare this report as well.

References

[1] Maymounkov, P. and Mazieres D. Kademlia: a Peer-to-Peer Information System Based on the XOR Metric. In *1st International Workshop on Peer-to-Peer Systems (IPTPS'02)*, (Cambridge, MA March 2002)

[2] Qin Lv and Sylvia Ratnasamy, *Can Heterogeneity Make Gnutella Scalable?* Proceedings of IPTPS 2002

[3] Ratnasamy, S., et al. A Scalable Content-Addressable Network. In *ACM SIGCOMM*. 2001. San Diego, CA, USA.

[4] Ratnasamy, S., et al. Location-Aware Overlay Construction and Server Selection. In *Infocom*. 2002.

[5] Rowstron, A. and P. Druschel. Pastry: Scalable, distributed object location and routing for largescale peer-to-peer systems. in *IFIP/ACM Middleware*. 2001. Heidelberg, Germany.

[6] Saroiu, S., Gummadi, K., and Gribble, S. A measurement study of peer-to-peer file sharing systems. In *Proceedings of Multimedia Conferencing and Networking* (San Jose, Jan. 2002)

[7] Stoica, I., et al. *Chord:* A scalable peer-to-peer lookup service for Internet applications. In *ACM SIGCOMM*. 2001. San Diego, CA, USA.

[8] Van Renesse, Robert and Birman Kenneth. *Scalable Management and Data Mining using Astrolabe.* Proceedings of IPTPS 2002.

[9] Xu, Zhichen and Zhang, Zheng, *Building Low-maintenance Expressways for P2P Systems*, available at http://www.hpl.hp.com/techreports/2002/HPL-2002-41.html, March 2002

[10] Zhang, Zheng, Shi, Shu-ming and Zhu, Jing. *Self-balanced P2P expressway*: *when Marxism meets Confucian*. MSR-TR-2002-72

[11] Zhang, Zheng, Shi, Shu-ming and Zhu, Jing. *SOMO: Self-Organized Metadata Overlay and its Application.* MSR-TR-2002-105

[12] Zhao, B., Kubiatowicz, J.D., and Josep, A.D. *Tapestry: An infrastructure for fault-tolerant wide-area location and routing*. Tech. Rep. UCB/CSD-01-1141, UC Berkeley, EECS, 2001.