Design and Implementation of Efficient Range Query over DHT Services

Xinuo Chen, Stephen A. Jarvis
Department of Computer Science, University of Warwick, Coventry, U.K.
Email: {xinuo.chen, saj}@dcs.warwick.ac.uk

***Abstract*-This paper describes the design and implementation of *DAST*, a Distributed Arbitrary Segment Tree structure that gives support of range query for public Distributed Hash Table (DHT) services. DAST does not modify the underlying DHT infrastructure, instead it utilises the scalability and robustness of DHT while providing simplicity of implementation and deployment for applications. Compared with traditional segment trees, the arbitrary segment tree used by a DAST reduces the number of key-space segments that need to be maintained, which in turn results in fewer query operations and lower overheads. Moreover, considering that range queries often contain redundant entries that the clients do not need, we introduce the concept of *Accuracy of Results* (*AoR*) for range queries. We demonstrate that by adjusting *AoR*, the DHT operational overhead can be improved. DAST is implemented on a well-known public DHT service (OpenDHT) and validation through experimentation and supporting simulation is performed. The results demonstrate the effectiveness of DAST over exiting methods.**

I. INTRODUCTION

There has been considerable research interest into Distributed Hash Tables (DHTs) in recent years. In addition to offering the advantages of scalability, load balancing and robustness, DHTs allow P2P applications to achieve efficient key insertion, lookup and retrieval over the underlying P2P network [1-4]. Imperative to the success of DHTs is the hashing operation. Each DHT node has a unique node identifier represented with a predetermined number of bits, e.g., a Pastry node has a 128-bit id [2]. The node identifier typically derives from the hash of the node's public key or IP address, and the set of node identifiers is uniformly distributed. Before inserting a key into the P2P overlay, DHT also hashes the key over the node identifier space so as to locate the node whose node ID is closest to the hash of the key. Once this mapping is complete, the hash of the key together with the value is stored at the target node.

The ID-based hashing effectively balances the load over all DHT nodes; however, this exact matching mechanism makes range query inefficient because clients can only search and retrieve one key at a time. If clients need to search for all available keys in a certain range, i.e., a *range query*, this is difficult to achieve via DHT lookup directly, since the DHT hashes the keys over the node identifier space before inserting, and the structural attributes of keys, such as the continuity of the key space, are erased by the DHT hashing functions. Consider for example that the keys to be inserted are the integers between 0 and 15. Each key is hashed before it is inserted into the DHT. If clients want to retrieve all keys in the range [3, 5], each key ("3", "4", "5") must be separately identified as even if one key is found, e.g., "4", it is not possible to conjecture the locations of its neighbours ("3", "5") through the hash value of "4" since the hashing is purely random and not structured. If the length of the range is very large, e.g. $[2, 2^{10}]$, then clients have to carry out 2^{20}-1 retrieval operations to obtain all keys, which introduces considerable overheads to the DHT [5] and the efficiency of the query itself.

To enable DHTs to support efficient range queries, we propose a Distributed Arbitrary Segment Tree (DAST), a data structure that is layered upon a traditional DHT. There exist a number of approaches to implementing a range query. In some designs keys are duplicated or the query results contain unnecessary keys in the interest of query efficiency. Nevertheless, the values associated with the keys are ignored. We believe that the size and type of the data associated with each key is crucial in understanding the efficiency of the query process. It is this data after all which is directly retrieved from the DHT and thus it is this that causes the storage load on the DHT. By considering the values associated with each key, DAST achieves a better balance between load and query performance. Moreover, we use the term *data item* of the form {*key*, *value*} when we describe DAST operations.

DAST constructs an arbitrary segment tree (*AST*), which is an enhanced form of a traditional segment tree [5, 6], to break down the entire key space into a number of segments (each segment being a node in the tree). For every insertion request of a data item {*key*, *value*}, DAST first locates all segments of the tree that contain the key, and then creates new data items in the form {*segmentId*, (*key*, *value*)}, i.e., DAST encapsulates the key and value in the new data item, with *segmentId* being the new key. Finally, DAST inserts the new data items into the underlying DHT instead of the original data items. To process a range query, DAST looks for a minimum number of segments on the tree so that the union of the selected segments matches the range of the query. This way, by retrieving all *segmentId*s in the union, we obtain the result of the range query. Since every segment contains a number of keys, retrieving by *segmentId*s instead of the original keys can significantly reduce the number of DHT retrieval operations and consequently improve the efficiency of the range query.

A novel concept in DAST is the accuracy of the results for a range query. As mentioned, the efficiency of DAST is determined by the number of segments that constitute the query range. The use of the arbitrary segment tree guarantees that the DAST is able to find the union of segments exactly matching the range. However, if we relax the requirement of

an exact match, that is, allow the union of segments to exceed the range of a query for a certain length, then fewer segments may be needed to cover the range, which in turn leads to fewer "get" operations to the DHT. This said, the query efficiency may not always be improved since the result of the query may contain unwanted data items due to the extra span of segments which may cause more traffic or longer latency. We thus define the accuracy of results (*AoR*) as the number of necessary keys divided by the total number of keys in the response. We analyse the balance between the efficiency of DAST and the value of *AoR* in this paper. To the best of our knowledge, no existing research has introduced or analysed the *AoR*, which makes our contribution unique.

Significantly, our solution does not require modifications to the core of the DHT; instead, we layer the DAST over a DHT infrastructure and present it as a middleware component between clients and DHTs. As some DHT systems have already become public services [7], this layering approach brings simplicity of implementation and deployment to applications. Note that DAST is a tree-based data structure, however, it does not require peers in the network to be organised in any particular overlay structure, i.e., the DAST tree does not require maintenance as long as the range of the key space is determined. Section III describes the characteristics of DAST in more detail.

The rest of the paper is organised as follows. We describe related work and compare DAST with this work in Section II. In section III we present the details of the DAST algorithms and the concept of the *AoR*. We evaluate the performance of DAST in Section IV. Finally we conclude in Section V.

II. Related Work

Range queries are used by many P2P applications, including P2P databases, distributed computing, and file sharing [8-10]. A variety of solutions have been proposed to address the range query problem for DHTs. These solutions can be classified into two broad categories: those that need to modify the core of the DHT, and those solutions that need not.

Mercury [11], SkipGraph [12], SkipNet [13], and PIER [14] are all representative examples from the first category. They either modify or redesign the core of the DHT to achieve a range query. Alternative designs include the Prefix Hash Tree (*PHT*) [15] and the Distributed Segment Tree (*DST*) [5], which represent examples of the second category, and subsequently do not need to know the internal mechanism of the DHT. Due to space limitations, we describe two examples, PHT and DST, and compare these with our own scheme DAST.

A. Prefix Hash Tree (PHT)

PHT employs a trie-based tree structure encapsulating the original tuples {key, value} in new data items with the label of the leaf nodes acting as the new key and inserting it into the underlying DHT. Each original key is expressed as a binary string of length *D*. All keys with the same prefix are stored on the same leaf nodes. The depth of the tree is decided by the load balancing mechanism in PHT, i.e., if the number of keys that are stored on a leaf node exceeds a threshold, the leaf node will split into two child leaf nodes.

Clients are not aware of the structure of the whole PHT. To determine which leaf node to insert, clients have to first look up all *D* possible prefix labels in parallel, e.g., if the binary string of a key is "00100", a client has to perform parallel "get" operations to the DHT for the keys "0", "00", "001", "0010" and "00100"; if one of the "get" operations returns a result, then the leaf node is located and the key is stored on it via the a data item. The authors of PHT also suggests a binary search solution for locating the leaf node [15]. For the query of range (L, H), PHT first locates the PHT node corresponding to the longest common prefix of L and H and then performs a parallel traversal of its subtree to retrieve all the desired data items as the result of the query.

DAST differs from PHT in the following ways. First, the depth of the PHT grows with the increase in inserted keys, i.e., the structure of PHT keep changing over time and as a result it additional "get" operations are required for each insertion operation. In contrast, the structure of DAST is stable as long as the entire key space does not change. Clients locate the destination tree nodes for keys without any additional "get" operations, which results in lower latency for range queries. Moreover, as will be described in Section III, the result of a range query in PHT may contain unnecessary data items, which may increase the latency. In comparison, DAST gives criteria for the accuracy of results. We find that with similar *AoR*, DAST requires fewer DHT operations and thus achieves lower latency for range query than PHT.

B. Distributed Segment Tree (DST)

The Distributed Segment Tree approach is the most similar to our work. Both DST and DAST use the concept of a segment tree [6], nevertheless, DST is a binary tree while DAST is multi-way. Each non-leaf node in DST has two children and the segment corresponding to the parent is split into two equal parts and assigned to the two children, respectively. Hence the entire key space is split into 2^i (i represents the level in the tree, counting from 0) parts on each tree level and the depth of the tree is $O(\log R)$ (R is the length of the entire space range). Therefore, keys need to be inserted to $O(\log R)$ DST nodes and there will thus be $O(\log R)$ duplications for each key (the number of duplications is not always $O(\log R)$ due to DST's load balancing mechanism which we describe in Section III). The nodes in a DAST can have more than two children and through setting the maximum number (M) of children that each node can have, there will be arbitrary number of segments on each tree level (this is where the name "arbitrary segment tree" derives) and the depth of the tree is $O(\frac{\log R}{\log M})$. Consequently, each data items in a DAST will have $O(\frac{\log R}{\log M})$ duplications, which leads to lower DHT storage load and operational overheads. DAST also adopts a load balancing algorithm that achieves similar effects to the one in DST, but with a considerably simpler implementation. Finally, DAST incorporates the concept of the *AoR* to further improve

the range query performance. DAST also provides clients with the flexibility to adjust the primary properties to suit their own range query requirements. Such an approach is not documented in DST or PHT.

III. DESIGN OF DAST

In this section we present the design of DAST. We first introduce the Arbitrary Segment Tree data structure and then describe how to layer an AST over an existing DHT infrastructure to achieve range query functionality.

A. Arbitrary Segment Tree

The Arbitrary Segment Tree (*AST*) is based on the traditional segment tree (*TST*) data structure [6], where a range (henceforth we use the term *segment tree range* to distinguish from the range in a query) of non-negative integers[1] is iteratively split at each level into certain number of segments, and each segment is assigned to one tree node. However, the rule of splitting the segment tree range on each level in AST is different from that found in TST. TST is a binary tree where every internal node has two children. Therefore, starting from the tree root, the segment that every internal node represents is evenly split into two parts and allocated to the two children, respectively, until it has only one number within. In contrast, AST is a multiway tree in which each internal node can have an arbitrary number[2] of children. We denote M as the maximum number of children that one node can have, i.e., each AST node can have at most M children. Note that AST is a superset of TST, i.e., when the value of M is 2, an AST becomes a TST. At each tree level, AST splits the segment tree range uniformly to up to M segments while maximising the interval size of each segment. The properties of AST are as follows:

1. Assuming the length of the segment tree range is R, the height of an AST is $O(\frac{\log R}{\log M})$.

2. The root node has the entire segment tree range. Every other node represents a segment. The union of all segments on the same tree level is the segment tree range.

3. Every non-leaf node has C_i children, where $C_i \le M$ and $C_i \neq 1$. The segment of each non-leaf node is split into C_i parts and distributed to the children. The value of C_i and the intervals of the segments for the children are decided by the tree construction algorithm (Algorithm I).

4. Every leaf node has an atom segment, i.e., a segment that contains only one key. The union of all leaf nodes covers the segment tree range.

5. Every node has a *segmentId*. DAST produces the *segmentId* by hashing its interval over the underlying DHT node ID space. Through the hash, the *segmentId* can be mapped to the DHT node ID space and then used in the DAST operations.

[1] The range that a segment tree represents can in fact include real numbers. In this paper, we only give examples of non-negative integers for practical purposes.

[2] The number cannot be "1" because splitting a segment cannot be performed if a node has only one child.

Algorithm I
The pseudo code of
the AST construction algorithm

```
// Parameters:
// ASTNode: the class of AST nodes.
// sf, st: bounds of the interval for the segment on the node.
// level: the tree level of the node
//ASTNode.children[]: an AST node's children.
//M: the maximum number of children; a global value.
//C: the number of children of a node

ASTNode(sf, st, level)
    C ← 0
    children ← new ASTNode[M]
    if sf ≠ st then
        from ← sf
        length ← (st - sf) / M
        to ← from + length
        while true do
            children[C] ← new ASTNode(from, to, level+1)
            C ← C + 1
            if to = st then
                break
            else
                from ← to + 1
            if st > from + length then
                    to ← st
            else
                    to ← from + length
```

Unlike PHT, an AST will not change its structure once it has been constructed, as long as the segment tree range does not change. This property ensures the consistency of the positions for keys, i.e., the destination node that holds the (key, value) items. Fig. 1 depicts an example AST where the segment tree range is [0, 16] and the value of M is four. Note that the numbers of children of internal nodes vary between two and four through the tree and are purely determined by the segment tree range and the choice of the value of M.

B. DAST operations

The DAST data structure provides an interface between the client applications and the underlying DHT. Clients insert, delete or retrieve data items to or from DAST instead of DHT. We describe the DAST operations needed to achieve range query functionality for clients.

1) *Insert/Delete*: The insertion and deletion of a data item with a key in DAST is straightforward. When an insert request arrives, DAST looks for all nodes whose segments cover the key of the item (there must be one and only one such node on each tree level). For each of these nodes, DAST creates a new data item in which the key is the *segmentId* of the node and the value is the original data item. Finally, DAST inserts the new data items to DHT. The insert operation for one key in DAST needs $O(\frac{\log R}{\log M})$ DHT insertions and there will be $O(\frac{\log R}{\log M})$ copies of the key inside the DHT. When a data item is deleted, DAST finds all segments that cover the key and removes the data items accordingly.

2) *Range query*: DAST first divides the range of the query into a union of segments that the AST contains, and then retrieves all *segmentIds* with associated data items from the DHT. The dividing algorithm is shown in Algorithm II. Since

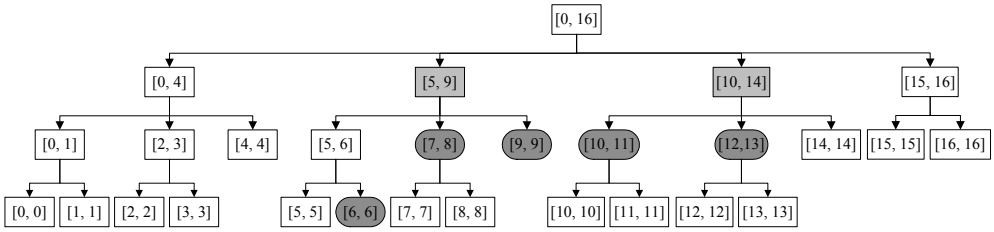


Figure 1: An example AST with the segment tree range [0, 16] and $M = 4$. We choose the segment tree range such hat each node can have an arbitrary number of children and the segments are uniformly split in each level while maintaining appropriate span length. An exemplar query for range [6, 13] is also illustrated here. The query union can be {[6, 6], [7, 8], [9, 9], [10, 11], [12, 13]} with *AoR* 100% or be {[5, 9], [10, 14]} with *AoR* 71.4%.

the AST ensures leaf nodes have atom segments, the union of the segments is guaranteed to be found for the range. There may exist alternative ways to divide the range; however, our algorithm is dedicated to building a union containing a minimum number of segments, i.e., the intervals of the segments should be as wide as possible, so as to reduce the number of DHT retrievals.

3) *Single key query*: DAST performs single key queries by simply retrieving the corresponding atom segment from the DHT.

C. The value of M

The value M controls the maximum number of children an AST node can have. The key advantage that AST has over TST is that it provides more flexibility for clients to improve the performance of a range query. As previously described, the height of an AST is $O(\frac{\log R}{\log M})$ and hence a greater value of M leads to lower numbers of DHT insertions (improving performance of the DAST insertion) and less duplications of data items (reducing the DHT storage load). However, if M is too large, the segment of one node will be split into more parts and consequently the segments in the AST will be shorter. Therefore, when fulfilling a range query, the average number of segments in the union that covers the range will be greater. In other words, DAST has to perform more DHT retrievals to obtain the result. Due to this tradeoff, clients have to carefully choose the value of M depending on their definition of the key space and their expectations for the lengths of the ranges that the queries may have. We investigate the impact of M on the performance of DAST in section IV.

D. Accuracy of Result for a range query

We consider the Accuracy of Result (*AoR*) for a range query in DAST. This investigation is motivated by the fact that when using PHT we found that the responses of range queries may contain unnecessary data items, since one prefix tree node stands for a prefix of keys and consequently keys that do not belong to the same range may fall into one prefix node. This causes higher latency to the query responses and cannot be rectified because PHT does not modify the DHT layer and so cannot filter the query results before feeding them back to the clients. By default, DAST always returns the query results to clients with 100% accuracy, i.e., the responses of the query do not contain any unwanted data items. However, we found that if we relax the segment union for the query (to be larger than the range of the query), i.e., the span of the union covers the range but has extra intervals on either end or both ends, the number of segments in the union may be reduced. Consequently, a number of unnecessary data items will exist in the results, however, the number of DHT retrievals needed for range queries will also drop. An exemplar range query [6, 13] is illustrated in Fig. 1. DAST builds a union {(6, 6), (7, 8), (9, 9), (10, 11), (12, 13)} for the query [6, 13] by default and has to perform five DHT retrievals for the

Algorithm II
The pseudo code of the dividing algorithm
for the range of the query

```
// Parameters:
// rf, rt: bounds of the interval of query range
// cdt: the candidate segment for the union of range segments.
// newCdt: new candidate segment.
// cdtClt: the collection of candidates (cdt).
// cf, ct: bounds of the interval for the candidate segment.
// nf, nt: bounds of the interval for the current AST node.
//nri: number of redundant data items allowed in query results.
// results: the union of segments that match the range.

divideRange(rf, rt, AoR)
    cdtClt.add(interval(rf, rt))
    for each level on the tree do
        for each AST node on the level do
            if candidates is empty then
                return results
        else
            for each cdt in cdtClt do
                nri ← (cdt.to ← cdt.from) × (1-AoR)
                newCdt ← interval(cdt.from-nri, cdt.to + nri)
                if newCdt covers the current node then
                    results.add(the segment of current node)
                    if cf < nf then
                        cdtClt.add(interval(cf, nf-1))
                    if ct>nt then
                        cdtClt.add(interval(nt+1, ct))
                    cdtClt.remove(cdt)
                    break
    return results
```

result. If we relax the union construction to be {(5, 9), (10, 14)}, the result may contain only two extra items (5 and 14) but the number of retrievals drops down from five to two, which is 2.5 times lower than before.

Achieving a range query in DAST usually requires a number of DHT retrievals and these DHT retrievals are executed in parallel which significantly reduces the response latency. However, if clients submit range query requests to DAST simultaneously with high frequency, DAST has to in turn submit the retrieval operations for those range queries to the underlying DHT in parallel and the DHT may suffer high overheads in a short period of time (PHT also considers the overhead for a DHT when choosing a binary search or parallel search for a lookup, although there is no detailed analysis in the associated paper). To help the DAST clients reduce the overhead imposed on the DHT, we present the concept of the accuracy of result (*AoR*) for a range query. We will show that by adjusting the value of *AoR*, the number of DHT retrievals for range queries can be much reduced and the overhead on DHT can therefore be lowered. The *AoR* is defined as the number of necessary data items divided by the total number of data items in the result of a query. In the example above, the value of *AoR* is $\frac{5}{5+2} = 71.4\%$ after tolerating unnecessary items in the result. The implementation of *AoR* is demonstrated in Algorithm II.

The *AoR* in a DAST range query is 100% by default since DAST builds a segment union that can precisely match the range of the query and the resulting response consists of only necessary data items. Clients can choose the desired *AoR* value to be less than 100% to suite their application environments. Note that the desired *AoR* acts as a threshold in DAST, i.e., the actual *AoR* of range query may not precisely equal the desired one but it is guaranteed not to be lower. This is because we assume every key in the key space as having a data item in Algorithm II, and calculate the *AoR* by the number of key slots not the number of actual items. In real range query cases, since some key slots may be empty, the actual *AoR* must be equal to or greater than the desired one. We demonstrate the relationship between *AoR* and the number of DHT retrievals in section IV.

E. Load Balancing

Approaches based on segment trees have potential problems on load balancing. There are fewer nodes at the higher tree levels; however, these nodes are responsible for more data items, as each data item has to be inserted into every tree level. The extreme case occurs at the root node. Since the root node has the entire key space, it will have to maintain a copy of every data item. The actual DHT node thus experiences a heavy storage load.

DST [5] employs a load balancing mechanism, called *downward load stripping*. Each node maintains two counters for its children, the left one and the right one. If, when a key is inserted into a node, it can also be covered by one of its children, the corresponding counter is increased by one. When either counter reaches a threshold, the node stops receiving keys. What this mechanism actually does is to limit the high level nodes from having more data items than the threshold. However, it brings to the implementation the problem of how do clients locate the values of two counters for each DST node in such a distributed environment? The obvious solution is to put the counters into the underlying DHT as data items and let clients access them through specified keys. However, this solution will occupy extra DHT storage and the insertions or retrievals of the counters themselves take time. Consequently, concurrency or synchronisation problem may occur, e.g., one node may not stop receiving data items when it should, because the counters are not updated on time.

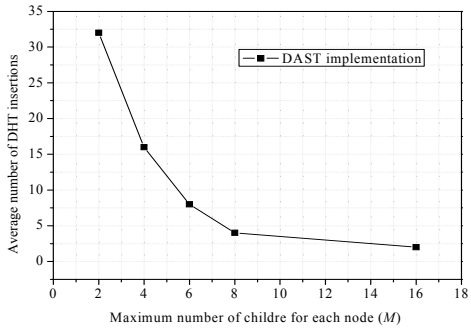
Load balancing is nontrivial in DHTs [16] and cannot be perfect since even if the keys are uniformly distributed onto the DHT nodes, some nodes will be responsible for a logarithmic factor more of the key space than others [3]. In other words, some nodes in the DHT will assume much higher storage and routing load than others. Due to inheritance, PHT, DST and DAST also suffer from the same problem. Even though data items are inserted at leaf nodes in PHT and it is easier to distribute leaf nodes uniformly unto DHT nodes, some data items within a certain range may still gain high popularity and become responsible nodes and hence will have uncharacteristically heavy load; this is also true for the DST.

Therefore, we propose to reduce the effects of load in DAST but not to perfectly eliminate it. We ignore the nodes in the levels N - 1 and above in the AST and start to insert data items at level N. The value of N depends on how large the entire segment tree range is and how many nodes there are in the underlying DHT. We encourage applications or clients to carry out experiments to test their values of N before deployment. Our evaluation in Section IV provides suggestions as to how to choose a good value for N.

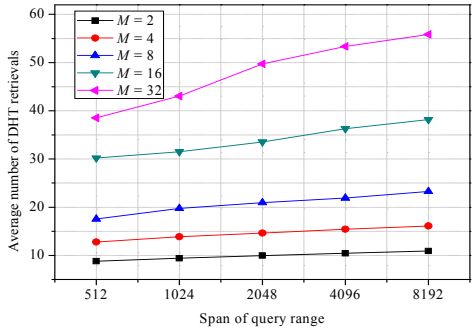
F. Tree Maintenance and Fault Tolerance

As previously described, DAST is a data-structure layer between the peer-to-peer overlay and the DHT infrastructure. When a peer carries out range query operations, it passes the command to DAST and its associated algorithms, and subsequently obtains the results from the DAST layer. Thus, the DAST data structure exists only within the application functions with which peers carry out range query operations; it does not influence the peer-to-peer overlay structure or the DHT infrastructure. Moreover, algorithms I and II show that the DAST structure will remain constant as long as the range of the key space does not change. Thus, DAST does not require additional maintenance which significantly simplifies the supported applications.

Since DAST is built upon a DHT service layer, it inherits all the resilience and failure recovery properties of the underlying DHT. Although the DHT has methods to guarantee a certain level of data availability and fault tolerance [7], the DAST can still lose data if all replicas in the DHT fail. To avoid this catastrophic failure, DAST employs a soft state refreshing mechanism. Each data item that is inserted to the DHT through the DAST layer has a time-to-live (TTL) associated. Peers have to regularly update the data items against a TTL of seconds, otherwise, the data items are automatically deleted from the DHT. Hence, even if all replicas for a data

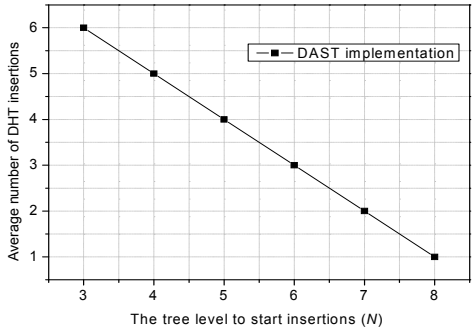


(a) The average number of DHT insertions for one DAST insertion

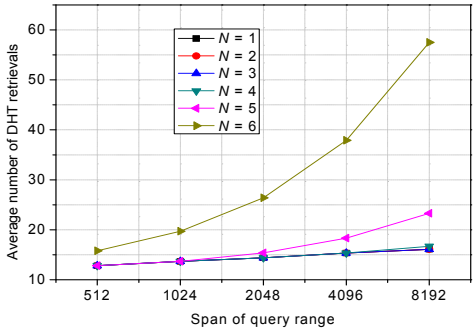


(b) The average numbers of DHT retrievals for one DAST range query

Figure 2: Plots of DHT operations for different values of *M* (Maximum number of children): (a) the plot of the average number of DHT insertions for one DAST insert request; (b) the plot of the average number of DHT retrievals for one DAST range query request



(a) The average number of DHT insertions for one DAST insertion



(b) The average numbers of DHT retrievals for one DAST range query

Figure 3: Plots of DHT operations for different values of *N* (the level number that DAST starts to insert data items): (a) the plot of the average number of DHT insertions for one DAST insert request; (b) the plot of the average number of DHT retrievals for one DAST range query request

item in the DHT are lost, the item will eventually be restored by the supporting refresh mechanism.

IV. EVALUATION

In this section, we evaluate the performance of DAST. First we investigate the internal structural properties of DAST. We then compare the range query operations of DAST, DST and PHT. Finally we compare their range query efficiencies in an OpenDHT deployment.

A. Implementation

We implement two versions of DAST, the first as a simulation and the second as a full-scale deployment. In both versions, the source codes for the mechanisms of DAST are exactly the same. The only difference is that the simulation version of DAST utilises a Java Hashtable object to simulate the underlying DHT, while the deployed version is layered on top of OpenDHT.

To shorten the time to conduct the experiments, we use the simulation version to investigate the structural properties of DAST and compare the range query operations of DAST, DST and PHT. For comparisons of the real range query efficiencies, we use our deployed version of DAST that accesses OpenDHT service on the Internet.

B. Setup

In the simulations, we assumed the segment tree range to be $[0, 2^{16}-1]$ and generated 2^{14} keys for insertions. The keys are uniformly distributed over the segment tree range space. The values associated with the keys are empty, i.e., the sizes of the values are zero. This is because the sizes of the data items do not affect the investigation of the internal mechanisms of DAST, DST or PHT - such a configuration also improves the simulation efficiency. We also randomly generate five sets of range queries, each of which has 1000 queries with span lengths of 512, 1024, 2048, 4096 and 8192, respectively.

In the deployment, the segment tree range remains the same but we generate only 2^{10} random keys. We chose a relatively small number of insertions because 2^{10} insertions are enough to demonstrate the insertion efficiency of all three approaches. Every key has 1KB of data associated with the value (the maximum size of a value in OpenDHT is 1KB). All the experiments were prototyped on a single PC to guarantee the correctness of the comparison of results. The range query

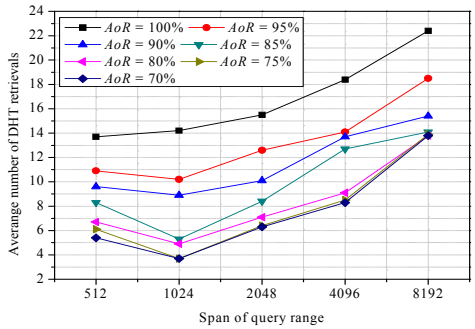


Figure 4: Plot of the average number of DHT retrievals for one DAST range query request with different values of *AoR* (the accuracy of result).

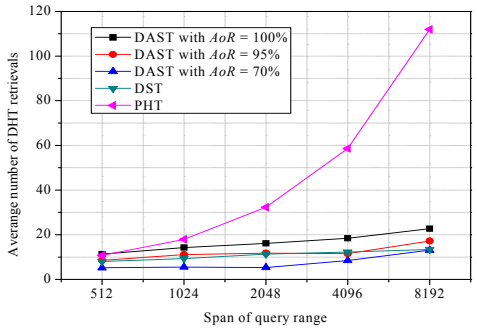


Figure 5: Comparison of DAST (with different *AoR*) against DST and PHT on average number of DHT retrievals for one range query.

setup is similar to that in the simulation except that each query set consists of 100 queries.

Each of the simulation experiments were conducted 100 times and the experiments on the OpenDHT deployment were repeated 30 times.

C. *Structural Properties of DAST*

We study the number of children allowed in AST, the load balancing mechanism and the performance impacts from different values of *AoR*. Clients can choose their own settings to suit the demands or adapt to the different computing environments.

Maximum number of children (the value of *M*): As described in section III, the value of *M* controls both the number of DHT insertions and the number of DHT retrievals for range queries. Recall that the height of AST is $O(\frac{\log R}{\log M})$, if *M* is too large, AST may have only a very small number of levels (the extreme case is that the whole AST has only the root node when *M* = *R)*. Thus to maintain the AST we choose the candidate *M* to be 2, 4, 8, 16 and 32. For each of the DAST examples with those *M* candidates, we insert the preloaded keys (for now we do not consider the load balancing problem and *AoR*) and plot the average number of DHT insertions involved. As depicted in Fig. 2(a), the number of DHT insertions drops sharply when *M* increases from 2 to 4 and this trend slows as *M* increases. When *M* reaches 16, the number of DHT insertions remains constant. To see how the value of *M* affects the range query, we send the five sets of predefined range queries to DAST and plot the results in Fig. 2(b). We can see that the higher the value of *M* leads to a larger number of DHT retrievals. The distance between the curves for *M* = 8 and *M* = 16 is large, indicating a sudden increase of DHT retrievals. Comparing Fig. 2(a) and (b), we thus suggest that *M* = 4is the optimal in our experiments.

Load balancing (the value of *N*): Our load balancing mechanism is simply that we start to insert data items from tree level *N* (if the root node is on level 1). The top *N* levels therefore contain no items. Using the result of *M* = 4 from the previous experiments, and testing *N* values from 1 to 6, the insertion and query range results are plotted in Fig. 3(a) and 3(b) respectively. The number of DHT insertions is reduced by one if *N* increases by one, which is apparent in Fig. 3(a). In

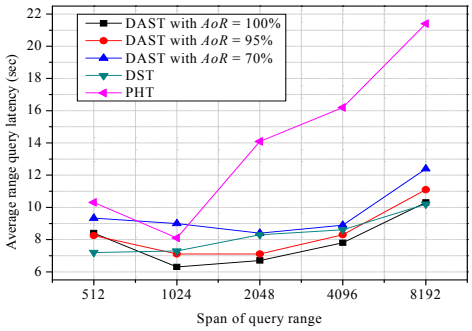


Figure 6: Comparison of DAST (with different *AoR*) against DST and PHT on query latency.

Table I
The experimental results for *N* = 4 and *N* = 5

N	# of nodes	Query span				
		512	1024	2048	4096	8192
4	64	12.8	13.7	14.4	15.35	16.7
5	256	12.8	13.7	15.4	18.4	23.3

Fig. 3(b), it is not easy to see the plots where *N* = 1, 2, 3 because they are overridden by *N* = 4, which implies the results for these three values of *N* are similar. When *N* = 5, the number of DHT retrievals starts to rise as the nodes on levels 1 to 4 do not have data items and cannot contribute to the range query. The increment is more pronounced when the value of *N* reaches 6. The results in Fig. 3(b) narrow our choice of *N* down to 4 or 5. We do not consider $N \leq 3$ because *N* = 4 gives better load balancing while providing a similar number of DHT retrievals. We present the detailed experimental results for *N* = 4 and *N* = 5 in Table I to illustrate choosing the optimal *N*. As we can see, the differences between the numbers of DHT retrievals of the two cases become larger when the span of the query increases. However, we should also note that the number of nodes on tree level 5 is 256, which is four times more than that on level 4. Considering that lowering *AoR* in DAST can further reduce the number of DHT retrievals, we thus chose 5 as the optimal value for *N*. This conclusion is validated in the next experiments considering *AoR*.

Table II
The experimental results for *AoR*

AoR	Average # of retrievals	Dropping percentiles for *AoR*	Dropping percentiles for # of retrievals
100%	16.84	N/A	N/A
95%	13.26	5%	21.62%
90%	11.54	10%	31.78%
85%	9.76	15%	43.19%
80%	8.32	20%	51.94%
75%	7.72	25%	55.98%
70%	7.5	30%	57.43%

Table III
The comparison of *AoR* between PHT and DAST

	PHT					DAST		
AoR	79%	86%	92%	96%	98%	70%	95%	100%
# of DHT retrievals	10.7	17.9	32.3	58.6	111.9	7.5	12.0	16.5

Table IV
The experimental results for the average latencies of insert and range query in DAST, DST and PHT

	DAST (M = 4, N = 5)			DST	PHT
	AoR =100%	*AoR*=95%	*AoR*=70%		
Insert (sec)	4.5			6.7	9.6
Query (sec)	7.9	8.37	9.606	8.32	15.88

Note, N = 5 is not universally optimal and clients should test for their own value of N.

The accuracy of the result for range query (*AoR*): To provide an analysis from the point of view of the *AoR*, we queried DAST for the same range sets seven times and each time we tested a different value of *AoR*. The value set of *AoR* are shown in Fig. 4. We do not present the results when *AoR* < 70% because these plots are masked by the plot for *AoR* = 70%, which means the value of *AoR* stops affecting DAST when it is below 70%. As we can see in Fig. 4, the number of DHT retrievals needed for the range query drops along with the reduction of *AoR*. We confirm the precise percentile of the drop (compared to that when *AoR* = 100%) with the corresponding value of *AoR* in table II. Through comparisons, we can see that if we reduce the value of *AoR* by even 5%, the number of DHT retrievals drops significantly (by 21.62%). If we allow 30% of the result to be unnecessary (*AoR* = 70%), the number of retrievals drops further to 57.43%.

Clients should be aware that lowering the value of *AoR* can also affect the response latency of the query depending on the sizes of the data items. If the size of the data item is small in the client application and the frequency of the range query request is high, having *AoR* of 70% can result in an approximate 50% lower overhead to the underlying DHT and may not negatively affect the response latency. Even if the size of the data items is large and the frequency of the request is high, allowing *AoR* to be 95% is worth considering since it still results in over 20% lower overhead to the DHT. A detailed analysis of the tradeoffs among the data size, overhead and *AoR* is required; this is precluded in this study as the implementation and evaluation of DAST is done entirely on a third party DHT layer. The results here provide suggestions rather than quantitative conclusions for reducing the potential DHT overhead through adjusting *AoR* in DAST.

D. Range query operations in DAST, DST and PHT

We compare the number of DHT operations (insertions and retrievals) that are needed for range queries in DAST, DST and PHT. The parameter settings M = 4 and N = 5 are selected for DAST and a block size of 60 is chosen for DST and PHT, which means that on each of the DST and PHT nodes they can have at most 60 data items stored (these settings replicate those found in related literature [5]). We insert the same set of data items to DAST, DST and PHT, and execute range queries using the same query sets in each of the three approaches. In DAST however, we also conduct range query experiments for three different values (100%, 95%, and 70%) of *AoR*; these results can be found in Fig. 5.

For an insertion request of one data item, PHT always requires only one DHT insertion, however, it requires a number of DHT retrievals for the lookup of the leaf node. For PHT, we hence add the number of DHT retrievals for the lookup to the one DHT insertion and treat the sum as the number of DHT operations needed for one data item insert request. The simulation results indicate that the average numbers of DHT operations for one data item insert request are 5, 13, and 8, respectively for DAST, DST and PHT. DST requires on average 13 DHT insertions for one data item insert request and duplicates the data item 13 times in the DHT storage. DAST requires less than half the DHT insertions and one data item requires only 5 copies in DHT, which significantly reduces the storage load in DHT. PHT on the other hand needs only one DHT insertion and requires only one copy of a data item. However, it requires on average 7 DHT retrievals, which imposes a higher operational overhead than DAST. To conclude, DAST is demonstrably superior to DST for insert requests and trades extra storage for insertion performance when compared to PHT.

Fig. 5 depicts the simulation results for the range queries. For one range query, PHT performs many more DHT retrievals than DAST and DST, which represents potentially high DHT overheads. When DAST is configured with *AoR* set to 100% it requires more DHT operations than DST. This is because each DST node has fewer children and the splitting of segments is slower than in DAST; DST therefore has longer segment spans, leading to fewer query unions of segments and fewer DHT retrievals. Nevertheless, when the *AoR* of DAST is set to 95%, DAST achieves approximately the same number of DHT retrievals as DST. When *AoR* is configured to 70%, DAST surpasses DST.

PHT does not always achieve 100% *AoR* in the results of the range queries. We calculate the *AoR* and the average number of DHT retrievals for PHT and DAST responses, and present the results in Table III. Through comparing the values of *AoR* in PHT and DAST together with the average number

of DHT retrievals, we can see that DAST performs fewer retrievals while maintaining higher *AoR*.

E. Comparison of the latencies for insertions and range queries in DAST, DST and PHT

In this experiment, we deploy our DAST implementation on OpenDHT together with DST and PHT. We insert the preloaded data items into OpenDHT through DAST, DST and PHT, respectively. The latency of every insertion is recorded and the average of these values is presented in Table IV. The results clearly indicate one DAST insertion takes on average only 67% of the time that DST insertion requires. The advantage of DAST over PHT is more pronounced in that PHT insertions take twice as long as DAST insertions.

For the range query experiment, we deploy three versions of DAST, each of which is configured with *AoR* as 100%, 95% and 70%, respectively. With different values of *AoR*, we investigate the impact of *AoR* on the query latency. These results are presented in Fig. 6; the average latencies of the range queries can be found in Table IV. We can see that the average latency in DAST with *AoR* as 100% is very close to the one in DST. When *AoR* is reduced, the latency grows due to extra unwanted items in the results. PHT requires more time for range queries because it needs several sequential steps to lookup the leaf key, and the response contains unnecessary items. DAST does not have sequential operations and thus performs better.

V. CONCLUSIONS

In this paper, we proposed a Distributed Arbitrary Segment Tree (DAST), a structure built on top of public DHT services to achieve enhanced range query functionality for clients. DAST incorporates the Arbitrary Segment Tree (AST), yet is designed so that the query union contains a smaller number of segments leading to fewer DHT operations and a lower overhead. In addition, the duplications of data items are significantly reduced in DAST as compared with DST. Moreover DAST introduces the concept of *AoR* (Accuracy of Result). By adjusting the value of *AoR*, we demonstrate that DAST can further reduce the number of DHT operations and therefore further reduce the overhead.

An advantage of this scheme is that DAST does not modify the underlying DHT and instead acts as a middle layer between DHT and the applications that require range query functionality. The approach is also designed to provide DAST users with the flexibility to modify DAST to their application environments for best range query efficiency. Furthermore, the DAST structure is deterministic once the range of the key space is decided. This is significant in terms of lack of maintenance, which itself simplifies and reduces the overhead to the supported client applications.

Validations are undertaken through both simulation and extensive real-world experimentation and the results demonstrate the effectiveness of DAST across a range of metrics.

ACKNOWLEDGEMENTS

This work is funded in part by the UK Engineering and Physical Sciences Research Council (EPSRC) contract number EP/F000936/1.

REFERENCES

[1] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A Scalable Content-Addressable Network," presented at ACM SIGCOMM, 2001.

[2] A. Rowstron and P. Druschel, "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems," presented at 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001), Nov. 2001.

[3] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan, "Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications," presented at ACM SIGCOMM, 2001.

[4] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz, "Tapestry: A Resilient Global-scale Overlay for Service Deployment " *IEEE Journal on Selected Areas in Communications*, 2003.

[5] C. Zheng, G. Shen, S. Li, and S. Shenker, "Distributed Segment Tree: Support of Range Query and Cover Query over DHT," presented at IPTPS, California, USA, 2006.

[6] M. d. Berg, M. v. Kreveld, M. Overmars, and O. Schwarzkopf, *Computational Geometry: Algorithms and Applications*, 2nd ed: Springer-Verlag, 2000.

[7] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu, "OpenDHT: A Public DHT Service and Its Uses," presented at ACM SIGCOMM, 2005.

[8] V. Papadimos, D. Maier, and K. Tufte, "Distributed Query Processing and Catalogs for Peer-to-Peer Systems," in *The Conference on Innovative Data Systems Research* Asilomar, CA, USA, 2003.

[9] M. Abdallah and H. C. Le, "Scalable Range Query Processing for Large-Scale Distributed Database Applications " presented at Parallel and Distributed Computing Systems, Phoenix, AZ, USA, 2005.

[10] D. Oppenheimer, J. Albrecht, D. Patterson, and A. Vahdat, "Design and Implementation Tradeoffs for Wide-Area Resource Discovery " presented at HPDC, 2005.

[11] A. R. Bharambe, M. Agrawal, and S. Seshan, "Mercury: Supporting Scalable Multi-Attribute Range Queries," presented at ACM SIGCOMM, 2004.

[12] J. Aspnes and G. Shah, "Skip Graphs," presented at ACM - SIAM Symposium on Discrete Algorithms (SODA), 2003.

[13] N. J. A. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman, "SkipNet: A Scalable Overlay Network with Practical Locality Properties," presented at Fourth USENIX Symposium on Internet Technologies and Systems, 2003.

[14] R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica, "Querying the Internet with PIER," presented at 19th International Conference on Very Large Databases (VLDB), 2003.

[15] Y. Chawathe, S. Ramabhadran, S. Ratnasamy, A. LaMarca, S. Shenker, and J. Hellerstein, "A Case Study in Building Layered DHT Applications," presented at ACM SIGCOMM, 2005.

[16] S. C. Rhea, "OpenDHT: A public DHT service," in *Computer Science*, vol. PhD. Berkeley: University of California, 2005.