A Survey of Distributed Data Structures in Peer-to-Peer Systems

Abstract. We present a survey of Distributed Data Structures (DDSs) in the Peer-to-Peer domain. We start from one of the earliest designs that were presented in the literature and build the survey from there. We try to classify the predominant DDSs in the literature and discuss their architecture, working, salient features and performance briefly. The paper is broadly divided into two categories: Hash-Based DDS and Order-Based DDS. We highlight the pros, cons, similarities and differences of both these categories and review some examples of each.

1. Introduction

In recent years there has been a substantial shift from centralized architectures to distributed and decentralized architectures. One such paradigm is the Peer-to-Peer (P2P) network. These networks have the characteristic that all nodes are equal and perform roughly the same functions to achieve a common goal. To address the underlying needs of these networks Litwin et al. [1] coined the term Scalable Distributed Data Structure (SDDS) and presented a design of the first prototype known as LH*. The defining characteristics of a SDDS as presented in [1] are as follows:

- No central directory
- Client information about the location of the data may be outdated and is adjusted only in response to queries
- Clients might end up sending the query to an incorrect node/server which will then be forwarded to the concerned node/server and evaluated accordingly causing the client information to be updated

Later the original LH* design was modified to cater for range queries in RP* [2]. Following along the same lines other researchers presented new and improved Distributed Data Structures that perhaps do not strictly adhere to the definition presented above. In this paper we present an evolutionary survey of these DDSs and discuss their underlying architecture and characteristics and highlight their performance in terms of time and space complexity. The paper broadly discusses two types of DDSs: the first type is hash-based DDSs and their associated hybrids and span sections 2-7 whereas the remaining sessions deal with the other type i.e. the order-based DDSs. In section 7 we review the hash-based DDSs and point out some relevant research questions that remain unanswered. A similar discussion is presented in section 12 for order-based DDSs in particular and DDSs in general.

2. Distributed Hash Table
In this section we present an overview of one of the earliest Distributed Data Structures: A Distributed Hash Table (DHT) as discussed in [3]. During the time this paper was written, DDSs were still in their infancy and consequently the design presented is preliminary and was later modified according to needs. The DHT is a self-managing storage layer that has been designed to run on clusters of computers such as p2p systems. It has its advantages over a Relational Database Management System [4] and a Distributed File System [5, 6] which are two other competing storage layers in the literature and present themselves as suitable alternatives.

2.1 Architecture

The DHT consists of the following 5 components

- CLIENT: A client can be any service-specific software that is running on one of the nodes in the system and communicates with the service instances in the cluster. An example of a client is a web-browser. In this architecture the clients are oblivious to the presence of a DDS in the system.
- SERVICE: A service can be roughly described as a set of cooperating software processes. Each of these processes is called a service instance.
- HASH TABLE API: The hash table API provides the interface between two other actors in the system namely: the service instances and the DDS Libraries.
- DDS LIBRARY: The DDS Lib is a Java-Based library that allows application to use the hash table API. It coordinates with the Bricks below to execute the functions of the API.
- BRICK: Bricks are responsible for actually housing the data in the system. Each brick comprises a chained hash table implementation with locks.

2.2 Partitioning, Replication and Consistency

The DHT is created by horizontally partitioning the table of data. We divide the partitions evenly across the bricks present in the system. The partitions are replicated over a subset of nodes in the cluster and we call each such subset a replica group. During node failure, data is still readily available on other surviving nodes from the failing node’s replica group. Similarly when new nodes join the cluster, the partitions are modified to maintain fair distribution of data across the brick nodes. The replication calls for a strict consistency mechanism which is achieved through an optimistic two-phased commit protocol that provides atomic updates to the hash table. Here the DDS Library acts as the coordinator in the protocol whereas the replicas function as the participants.

2.3 Functioning

We now summarize two of the most common functions of the DHT i.e. get() and put(). When a client issues a get() query for a particular key, the DDS library needs to find the specific
partitions dealing with the key and the associated replica group. To cater to these requirements the DDS library consults two metadata maps that are replicated on each node in the cluster. The data partitioning (DP) map takes as input the key and returns the key’s partition name (RG name) and the replica group (RG) map takes as input the name of the partition and returns the replica group in the form of a list of bricks. The DP map can be thought of as a trie over the alphabets of the key space whereas the RG map is simply a lookup table as shown in Figure 1 [3].

![Diagram of DP and RG maps]

Figure 1 DP map on the left and RG map on the right

The put() key function is done similarly. First the library calculates the correct partition the key should go to and then determines the replica group of the partition using the RG map. The library then performs the write using the two-phased commit protocol mentioned above.

In the next sections we further the concept of hashing and present modified designs of DHTs.

3. Consistent Hashing Based Structure

We explain the concept of consistent hashing in a distributed environment as discussed in the Chord protocol [7]. The defining characteristic of consistent hashing is that it balances the load uniformly across the network with high probability. This is a desirable property for p2p networks as it does not burden any single node or subset of nodes with “unfair” load. The primary objective of the chord protocol is that it takes a key and maps it to a node in the network.

3.1 Architecture
The chord architecture takes a node’s IP address and assigns it an m-bit identifier using the SHA-1 hash function \([8]\). Similarly, an m-bit identifier is produced by hashing the key. The nodes are ordered in a ring, based on the values of their identifiers modulo \(2^m\). Keys are assigned to the nodes by comparing the value of the key identifier with the node identifier. The first node with the identifier equal to or greater than the key identifier is responsible for storing the key-value pair. Furthermore, each node stores a table of pointers called the finger table. This table has the property that the \(i\)th entry in the table points to a node that is roughly \(2^{i-1}\) hops away. The finger table is used for efficiently routing queries to other nodes while keeping the storage overhead low (\(O(\log n)\) with high probability).

### 3.2 Searching

We highlight the search mechanism of the Chord protocol here. When a node wishes to query the network for \(k\) it calculates the identifier of \(k\) by hashing \(k\). It then compares it with the node identifiers in the finger table and routes the query to the node having the largest identifier smaller than \(k\) (maximum of all node identifiers that are less than \(k\)’s identifier). The same process continues at the node receiving the query. Since the distance between the initial node and target node is halved with each hop, the total nodes that must be contacted are \(O(\log n)\) with high probability.

### 3.3 Node Joins and Departures/Failures

Each node joining Chord has to undergo a three step process to establish itself as a legitimate node in the system. The three steps involve (1) initializing its own finger tables and pointers, (2) updating the finger tables of existing nodes and finally (3) transferring keys that were previously the responsibility of some other node to itself. The entire process takes \(O(\log^2 n)\) time with high probability and \(O(\log n)\) time after applying certain optimizations. The Chord protocol comes bundled with a basic stabilization protocol that is responsible for keeping the node pointers up to date in the event of node departures or failures. Each node maintains a successor-list which the stabilize routine uses if a failure occurs. In a nutshell, if a node observes that a successor has failed, it simply replaces that node with the first live entry in its successor list.

### 4. Viceroy

Viceroy \([9]\) is another DHT based DDS that furthers the design of Chord by introducing features of a butterfly network \([10]\). The design choices of Viceroy enable it to have \(O(1)\) linkage cost and routing with \(O(\log n)\) hops. We elaborate more on these in the following sections.

#### 4.1 Architecture
Since we have already defined the Chord infrastructure we discuss here how Viceroy approximates to a classical butterfly network. Each node has a total of 7 pointers along with a level $l$ chosen randomly from amongst $\log n$ levels. The 7 pointers are the following:

- 2 pointers are directed towards the immediate successor and predecessor of a node forming the general-ring
- 2 pointers are directed towards the immediate successor and predecessor at level $l$ forming the level-ring
- A “down-right” pointer to a node at level $l + 1$ which is at a distance of roughly $1/2^l$
- A “down-left” pointer to a node at level $l + 1$ which is close-by
- An “up” pointer to a node at level $l − 1$ which is close-by if $l > 1$

The last three pointers together form the butterfly. The level selection is done through an elegant strategy where each node first estimates the total number of nodes in the network by figuring out the number of hops between itself and the node with the next identifier ($n_0 = 1/calculated\ distance$). After this a level is selected randomly from the interval $[1,..,\log n_0]$ with uniform probability (level-1 is considered the highest level). The simplicity of this scheme entails that a node should only reselect a level if its immediate successor departs/fails.

4.2 Searching

A query is routed in three stages. In the first stage, using the “up” butterfly links, a query is first routed upwards to a node at level-1. In the second stage, using the “down-right” and “down-left” links, a query is routed downwards towards the target. Once on the same level as the target, routing is done using either the level-ring pointers or the general-ring pointer eventually arriving at the destination. With high probability, routing is done in $O(\log n)$ time. The proof builds upon sanity (no. of levels < 3 log n w.h.p) and goodness (deals with the density of servers), two must attributes for a network to support Viceroy and considers all three phases separately to arrive at $O(\log n)$.

4.3 Node Joins and Departures

The join and leave routines are quite similar to that of Chord. They both make use of the search routine described above and just like Chord involve phases where the nodes have to update their pointers and transfer data (key-value pairs) which they are now responsible for.

5. P-Grid

P-Grid [11] can be thought of as a virtual distributed search tree on top of a DHT like structure. The specific design principles, discussed below, enable the P-Grid system to self-organize the
overlay network similar to Gnutella [12] or Freenet [13] with a scalable architecture in terms of deployment.

5.1 Architecture

Each key begins with a binary prefix. Nodes are responsible for storing data items whose keys have the same prefix. A node finds its place in the tree by travelling down the tree following the prefix of its associated keys. The architecture resembles that of a binary trie where elaborating a key’s prefix leads us to the node responsible for storing the key. Furthermore, each peer keeps pointers to its corresponding sibling along its path. For example if a node’s path is 100 then it will keep pointers to nodes whose paths are 0 (sibling at level 1) and 11 (sibling at level 2) and 101 (sibling at level 3). This mechanism enables a node to route a query to another node if it cannot satisfy it e.g. if a node receives a key with prefix 101 and its own path is 100, the node forwards the key to its sibling with path 101.

5.2 Load Balancing and Replication

In P-Grid load balancing is achieved through pair-wise interactions between the nodes instead of consistent hashing. Depending on the state of the data within a key space, the participating nodes modify the node paths, either by extension or retraction, resulting in a change in the routing infrastructure and consequently the key distribution. The new shape of the virtual tree manifests itself in the underlying routing tables. Furthermore, replication can be achieved at the node paths by appointing numerous nodes to be responsible for the same key prefix. This replication again triggers load balancing and the self-organizing nature of P-Grid uniformly distributes data over the replicas and across the system.

5.3 Searching

Routing is carried out by reading one bit at a time, of a given key, and forwarding it on the corresponding path along the tree. With high probability, the entire process takes $O(\log n)$ time.

5.4 Updates

P-Grid handles updates in a decentralized manner and provides probabilistic guarantees on the consistency of the data. The update algorithm is based on randomized rumor spreading [14] and employs a decentralized push/pull based generic gossiping scheme tailor made for p2p systems.

6. P-Trees and G-Grid

Systems described previously lack the functionality to evaluate a range query on the data items. P-Tree [15] is a Distributed Index Structure that provides p2p systems with the ability to carry
out equality and range queries. The design incorporates the concept of a Distributed B+-Tree, which are primarily used in database systems for evaluating queries [16], on top of a successor-maintenance algorithm such as Chord. The P-Tree differs from the B+-Tree in two primary aspects: the former has overlapping ranges and same data values are accessible through multiple sub-trees. The resulting P-Trees support both equality and range queries.

Another important design worth mentioning here is the G-Grid [17]. Much like the P-Grid, the G-Grid has self-organization properties but has the added ability to evaluate multidimensional queries on data. The entire object space is partitioned into regions, represented by a 2-tuple (x,y), based on attribute values. One node in the tree represents one region and an edge from the father node to the child node indicates that the child region is a subset of the father region. Regions can be further split into smaller regions that are either disjoint or overlapping and can also be merged together when the need arises, hence the self-organization. The nodes can be categorized as either s-peers or c-peers with the s-peers being responsible for managing at least one region of the G-Grid. One interesting property of G-Grid is that as the peers communicate during routing they learn about new peers and iteratively build an internal map of the whole space resulting in improved communication times.

7. Discussion on Hashing-Based Structures

We provide a rudimentary classification of the structures presented above and discuss their pros, cons, similarities and their differences.

7.1 Hash-Based Structures:

With a few exceptions most of these systems require O(log n) links per node and O(log n) hops to route a package. Viceroy, discussed above, and Koorde [18] can route packages with the same complexity but do so with O(1) links per node. Most schemes here implement a similar ring-based arrangement but differ in the number of pointers stored at each node for routing. Apart from logarithmic time routing, hash-based structures also balance the load nicely, such as in consistent hashing. However, these hashing mechanisms tend to destroy the ordering on the keys and cause data to be scattered all over the system based on the outcome of the hash function hindering the ability to perform range queries. This could potentially result in data being stored far away from its frequent users. Most of these structures employ the use of expensive stabilizing, maintenance or recovery protocols that run in the background in the case of failures to achieve consistent system state. Furthermore, hash-based structures usually lack the ability to self-organize.

7.2 Hybrids with Virtual Overlays:
Structures, such as P-Grid and P-Trees, which base their routing on virtual overlays atop a DHT like structure do away with some weaknesses of the DHT. These systems have strong self-organizing features. Local interactions between peers results in a consistent global state. These systems tend to be more resilient to failures and usually involve a local operation to fix inconsistencies. Their search and update times are comparable to those of strict hash-based systems. However a fundamental weakness in these structures is the restricted range of the key space as very large keys are unsuitable for key comparisons and routing. Furthermore, load balancing is achieved through a separate mechanism that might require additional overhead.

The G-Grid design is more evolved in the sense that it addresses the weaknesses of most of the systems described above and introduces geometric optimizations to enhance structural integrity and functionality. It also incorporates the idea of “learning” in to the algorithm which speeds the process of routing.

An open problem is to consider these structures under the Byzantine Failure model. Little or no work has been done in the literature to cater for such a failure model. Assuming that all failures in the P2P domain are crash failures is an unrealistic assumption. Furthermore, it remains to show how these systems would perform if an adversary knew the topology of the network and caused specific nodes in the layout to fail. These systems take very little advantage of content and path locality of objects and employ no use of caching mechanisms. It would be interesting to see the performance of such systems under the aforementioned optimizations.

We now move on to a different category of DDSs that are all derived from the same data structure; a skip list [19].

8. Skip Lists

Skip Lists were first presented as an alternative to balanced trees such as Red-Black Trees and AVL Trees because of their simplicity. They can be modified very easily and as we will show later they have been adopted in numerous p2p designs. All operations in skip list are local and require lock acquisition on a small number of nodes.

8.1 Architecture

Skip lists can be thought of as “a tower of linked lists” in which each layer above has increasingly spaced out elements as compared to the layer below, with a head and a tail tower at the two ends. The bottom most layer is a normal sorted linked list. The first level linked list has pointers to every other element and skips one element. The second level linked list points to every fourth element and skips 3 elements and so on. A classic example of linked list can be found in Figure 2 [19]. It is easy to see that the total number of levels for such a construction are O(log n).
This however is a perfect skip list and requires substantial rearrangement of elements at the time of insertion and deletion. A randomized skip list, presented in Figure 3 [19], on the other hand does away with the rearrangement cost. We shall see the operations on a randomized skip list below and shall refer to it henceforth simply as a skip list.

8.2 Searching

Search is triggered at the top most layer of the head tower. At each point we can either go forward in the list or go down. This action is determined by comparing the next element in the current list with the target. If the target is less than the next element we go forward otherwise we jump down. We proceed in this manner “skipping” over as many nodes as we can without overshooting the target. The search routines takes $O(\log n)$ time with high probability. The proof is constructed by backtracking the path of a search trace and separately determining the probabilities of climbing up and moving left. A detailed probabilistic analysis can be found in [19].

8.3 Insertions and Deletions

To update a skip list we first perform a search followed by a *splice* operation. In the case of an insertion we first determine the place the element is supposed to go in and then insert the element there. The levels can be chosen by flipping a coin if the probability is $\frac{1}{2}$. During the search we keep a record of the largest element at each level which is smaller than the node to be inserted and establish forward pointers from these nodes to the target node during the splice operation. The delete operation is intuitive and simply removes the node adjusting the pointers accordingly. Both these operations require $O(\log n)$ with high probability.

9. Skip Graphs
The skip list structure allows search to be initiated at the head tower only. For this reason it cannot be deployed over a p2p network. Furthermore it is not resilient to node failures. We now present a modification to the skip list known as a Skip Graph [20] which has desirable features for an underlying p2p system. Since there is no hashing, related resources can be placed near each other giving skip graphs the ability to evaluate range queries.

9.1 Architecture

A skip graph is structurally quite similar to a skip list except that there are multiple doubly linked lists at each level $i$, where $0 < i < O(\log n)$ on average, instead of one single list. Each node is a member of exactly one of these lists. The fact that there are many lists at each level implies that the chance for a node participating in a search is small making it more resilient. Each node stores a membership vector with it. These vectors are used to determine which list a node belongs to at a particular level. Nodes that have the same prefix of length $i$ in their membership vector will be part of the same list at level $i$ as shown in Figure 4.

![Figure 4 Showing a Skip Graph with maximum level 2](image)

9.2 Searching

The search routine is exactly the same as a skip list with a minor difference. Instead of starting at the head tower the search starts at the top most level of the node that is initiating the search. The routine filters out only those lists that the starting node is part of giving us a simple skip list structure to search through. The search operation takes an expected $O(\log n)$ time and messages.

9.3 Insertions, Deletions and Failures
A range query is performed to find out the node that has the nearest key at level-0. From there we climb up finding the nearest node on each level $i$ that has a matching prefix of length $i+1$. Once such a node has been identified we simply join the list that node belongs to. The insert routine takes an expected $O(\log n)$ time and messages. The delete routine removes the node from the graph and adjusts the pointers accordingly. In the case of a failure a repair mechanism is triggered. The authors of [20] describe some constraints on the skip graph that are violated in the case of a failure. The repair mechanism probes the network and sends out repair messages to the violators returning the graph to a consistent state. The skip graph performs well under various failure models. For $f$ targeted failures brought forth by an adversary, $O(f \log n)$ nodes are separated from the primary component. Under a random failure model nearly all nodes remain connected to the primary component even when the probability of node failure exceeds 0.6.

10. SkipNet

A SkipNet [21] is constructed on the same principles as a skip list and closely resembles a skip graph. The difference between SkipNet and skip graph is that the former focuses more on content and path locality while still providing load distribution whereas the designers of skip graph focus more on its algorithms and formally characterize its invariants and properties. SkipNet employs the use of two separate address spaces and uses a combination of both for routing and load balancing.

10.1 Architecture

A key underlying design principle of SkipNet is to maintain a sorted list of data items with pointers that skip over varying number of data items similar to a skip list. The basic structure is a doubly-linked ring of nodes that is sorted on the basis of Name IDs. Each node in this ring stores $2 \log n$ pointers in a routing table called an R-table such that the $i^{th}$ level pointers point to nodes that are roughly $2^i$ hops away on either side of the node. Similar to the skip list discussed above, the authors here present a probabilistic SkipNet architecture. It is probabilistic in the sense that when a ring at level $i$ is split in to two sub-rings the node randomly decides which of the two rings at level $i+1$ it wants to be part of. Each node also has a Numeric ID with the property that a prefix of length $i$ of this ID determines which ring the node belongs to at level $i$. The complete architecture can be seen in Figure 5 [21].
Routing can proceed by one of two available means. We summarize both here. *Routing by Name ID* is similar to that of a skip list where a message is routed to the node that is farthest away without overshooting the destination. This is done by consulting the pointers in the R-table. Once no more “skipping” is possible the message is passed from neighbor to neighbor until it arrives at the destination. *Routing by Numeric ID* is performed by examining the prefix of length $i+1$ of the packets destination address and match it with a ring ID at level $i+1$. Once the packet has arrived at a ring such that none of the rings at level $j+1$ match the prefix of length $j+1$ then the node that has the numerically closest ID to the destination receives the packet. The time both these mechanisms take is of the order of $O(\log n)$ with high probability. A proof can be found in [22].

**10.3 Node Joins and Departures**

When a new node wishes to join the network it first finds a top level ring to go to which is equivalent to routing a packet to the newcomer’s Numeric ID. It then determines its neighbors and populates its R-Table using the *Routing by Name* routine. The time for a node to join the
network is $O(\log n)$ with high probability. Departures are handled by a background process similar to other systems presented earlier.

10.4 Content and Path Locality

An ingenious part of SkipNet’s design is that it provides content and path locality. DNS names are reversed (www.uiuc.com becomes com.uiuc.www) to arrive at nodes’ Name IDs. When storing a document on a node of our choice SkipNet simply appends the Name ID of the node before the content name e.g com.uiuc.www/doc_name. This produces content locality. Path locality is also achieved as a result of inverting the DNS names because nodes belonging to the same organization will map to the same Name ID prefix resulting in greater probability for path locality.

10.5 Constrained Load Balancing (CLB)

In SkipNet each data object’s name is split into two parts: A CLB domain which indicates that load balancing should occur over all nodes that have the given CLB domain and the CLB suffix which determines the particular node in the domain storing the data item.

11. Skip Tree, Family Tree and Skip-Web

We now present a brief overview of 4 DDSs that are built upon principles borrowed from the primary skip list data structure. This goes to show the “potential” of these lists particularly in the p2p domain.

A SkipTree [23] divides the space into regions where each node in the network is a leaf node of this partition tree and is responsible for storing a region. The leaf nodes together form a SkipNet with wraparound after the last leaf node towards the first leaf node. The design incorporates elegant geometric optimizations resulting in $O(\log n)$ hops for routing. Storing links requires $O(\log n \log \log n)$ on average and $O(\log^2 n)$ in the worst case.

Family Trees [24] are a mix of Viceroy and SkipNet. Each node has 9 pointers, 2 more than Viceroy, with the added difference that each node is now part of three different rings instead of 2. 3 rings amount to a total of 6 pointers (3 successor and 3 predecessor pointers) and the remaining 3 are analogous to the butterfly network. The family tree structure provides $O(\log n)$ search, insert and delete operations while maintaining $O(1)$ link storage at each node. It provides a certain degree of content locality resulting in the ability to evaluate range queries.

Skip-Web [25] is a very interesting framework. It discusses a structural property of data structures called a range-determined link structure and defines a lemma called the Set Halving Lemma. Skip-web proves that all range-determined link structures satisfy the set halving lemma which in turn qualifies these structures to be used with Skip-Web’s scheme. The authors in [25]
present examples for the construction of skip-webs over one dimensional data structures such as sorted linked lists and multidimensional data structures such as Quadtrees, Octrees, Tries and Trapezoidal Maps. Operations on skip-webs are comparable to their alternatives and perform better at times.

12. Discussion on “Skipping”/Order Based Structures

Structures such as Skip Graphs and SkipNet have high resilience to node failures even if an adversary were to target a specific set of nodes. Others, like the Skip-Web structure, take a structure and develop it in to a DDS with performance comparable to other more complicated DDSs. These structures utilize the ordering of data for routing and load balancing purposes. Most of them are constructed with the same design principles at heart for which reason their algorithms and functional time and space complexities are more or less identical. The most striking advantage of these DDSs over hash-based systems is that they provide content and path locality which is highly a desirable feature of contemporary p2p networks. Furthermore data ordering paves the way for range queries and orthogonally multidimensional queries. These structures also rely on repair mechanisms as the architecture gets more complicated however in the simple cases it merely involves rearranging the pointers.

Byzantine Fault models need to be applied to these designs as well to test them for reliability. Furthermore, the loopholes in the protocols need to be tested for various security breaches. An immediate danger for p2p systems is not that they are vulnerable to attacks themselves but the fact that the sheer number of these nodes can be used to launch a costly attack on other systems such as a DDoS attack. Another open question is that of secure pair-wise communication. Most p2p systems are developed with this underlying assumption however this does not seem a plausible assumption and needs to be addressed. Other issues deal with data authenticity and access control mechanisms in p2p protocols.

Peers may also wish to connect to only a subset of nodes such as their friends or within a certain commercial organization. Current p2p systems lack this functionality and provide little or no autonomy to the actual users. Another aspect is that of advanced querying capabilities such as relevance ranking, age of data etc. Optimizations based on caching, proxies etc need to be explored further to speed up existing p2p networks. All these issues require a mammoth effort on part of the p2p research community.

13. Conclusion

DDSs have come a long way. Simple legacy structures have evolved in to complex and sophisticated systems with improved performance and functionality. However, the problem space for p2p networks has also grown exponentially resulting in a substantial void that needs to be filled in swiftly. As this survey highlights, most DDSs in this domain have been a
refinement and modification of older and simpler versions of centralized data structures such as a hash table or a linked list. Current designs are sub-linear in their performance but lack resilience and robustness. Perhaps future endeavors should focus more on the security aspects of these protocols rather than improving performance. A good sign for this field is that ideas of one researcher find their way in some form into the designs of another colleague resulting in greater collaboration and understanding.

References:


