Distributed Hash Table Storage Model for Distributed Network System

Ogbasi, P.A.[1] & Eneh, A.[2]
Department of Computer Science
University of Nigeria
Nsukka, Nigeria
E-mails: ogbasiprogress@gmail.com[1], agwozie.eneh@unn.edu.ng[2]
Phone: +2348068568027

ABSTRACT

Distributed Hash Table (DHT) data structure provides a viable technique for consistent and easy data access and management capabilities to distributed platforms. In this work, we used DHT capabilities in analysing and designing a reusable, distributed network based storage system for implementation in distributed environment, called Unified Distributed Storage System *(UniDSS)*, and specifically designed for the needs of Internet services. A UniDSS presents an in-memory data structure interface to applications implementing a modified chord DHT algorithm, and durably manages the data behind this interface by distributing and replicating it across distributed nodes. This paper describes the design architecture, and implementation requirements of distributed data structure (a distributed hash table built in Java Hadoop platforms). We evaluate its performance, scalability and availability, and its ability to simplify service construction.

Keywords: DHT, Distributed Network, Nodes, finger/Identifiers, UniDSS.

ISTEAMS Cross-Border Conference Proceedings Paper Citation Format
Ogbasi, P.A. & Eneh, A. (2017): Distributed Hash Table Storage Model for Distributed Network System.
Proceedings of the 9th iSTEAMS Multidisciplinary Conference, University of Ghana, Legon, Accra Ghana. Pp 89-102.

1. INTRODUCTION

In recent time, Hash Table data structure (especially Distributed Hash Table – DHT) plays an important role in distributed systems and applications, like in large-scale distributed environments. In the normal Client/Server network architecture (C/S architecture) since the central server is in charge of most of the resources, it becomes the most important part as well as the bottleneck and weak point of the system [2]. On the contrary, the distributed network architecture (a typical one is the peer-to-peer - P2P architecture) distributes the resources on the nodes in the system. The distributed model provides better robustness and more efficiently utilizes all peers' capability, while the resources of the clients are idle in C/S mode.

One of the major rationales of distributed network storage systems is to store data reliably over long periods of time using a distributed collection of storage nodes which may be independently unreliable. Most applications uses storage in large data centers and peer-to-peer storage systems such as OceanStore [3], Total Recall [4], and some of these applications makes use of Distributed Hash Table data models [5]. They use nodes across the network (ideally across Internet) for distributed data storage. In distributed wireless sensor networks, obtaining reliable storage over unreliable nodes might be desirable for robust and resilient data recovery [6], especially in catastrophic scenarios [7]. In all these scenarios, ensuring reliability requires the introduction of redundancy. The simplest form of redundancy is replication, which is implemented in many modern distributed storage systems.

1.1 Statement of the Problem

In distributed environments a key problem is how to manage the aggregate storage resources on connected systems efficiently. This is a particular important issue in large-scale systems involving Big Data set for processing at different nodes in a distributed computing environment.

Implementation of DHT is a suitable solution to address the issue of nodes storage utilization in a distributed environment and to promote the development of efficient computing resources in a distributed network systems greatly. DHT capability is a suitable structure for the design of storage systems for distributed networks. Fine-tuning the functions of a DHT to deal with distributed data at various network nodes in a distributed environment, and also leverage on the fact that DHT does not necessarily requires a central server and hence its techniques treats all DHT nodes in the distributed system equally.

1.2 Aim and Objectives of the Research

The aim of the research is to use Hash Table data structure technique and algorithm to develop a system model for data storage, efficient management and utilization of storage resources in distributed network systems. In specific terms, the objectives include:

- To survey and explore different storage models, techniques and architectures for large data set.
- To efficiently manage storage locations to enhance scalability, availability and consistency of data on a distributed platforms (nodes).
- To design a reusable data storage system for distributed network platform.

2. REVIEW OF RELATED LITERATURE

In this section, we briefly review distributed network systems architecture, and then discuss a set of DHT variants, which widely influence the design and development of distributed storage systems in recent times. In each case, we first describe the structure of each variant, present the key elements such as the routing model and the data model, and then discuss the solutions to a common problem in distributed environments and its storage systems, namely, nodes dynamically join and leave distributed systems. In the last section, we compare the DHT variants from numerous aspects, such as overlay network topology, distance metric, routing and data model and so on.

2.2 Distributed Hash Table

A distributed hash table (DHT) is a class of a decentralized distributed data structure that provides a lookup service similar to hash table: (*key*, *value*) pairs are stored in a DHT, and any participating nodes can efficiently retrieve the value associated with a given key, [8]. Responsibility for maintaining the mapping from keys to values is distributed among the nodes, in such a way that a change in the set of participants causes a minimal amount of disruption. This allows a DHT to extend to extremely large numbers of nodes and to handle continual node arrivals, departures, and failures. DHTs form an infrastructure that can be used to build more complex services, such as Anycast, cooperative Web Caching, Distributed File Systems, Domain Name Services (like domain name lookup), instant messaging, multicast, and also Content Distribution Systems. Notable distributed networks that use DHTs include BitTorrent, Distributed Racker, the Coral Content Distribution Network, Kad network, Storm Botnet, Tox Iinstant Messenger, FreNet and YaCy search engine, [3]. DHTs were first concurrently proposed around 2001 by several different research groups (Chord from MIT, Pastry from Rice/Microsoft, CAN from ICSI, and Tapestry from Berkeley), which basically all had a similar flavour, [8]. They provide a distributed data structure that exposes two functions:

put(key, value) and ***get(key)***………… (1)

These proposals, as in [11] took load-balancing properties first considered by the consistent hashing work of Karger et al (1997), differing because consistent hashing had each node have a global view of the system (where all system participants (nodes) know all other nodes in the system) where DHTs focused on more scalable designs (where nodes only know a fraction of total system participants).

For most of these algorithms, this "fraction" is O(log n), although you now see DHT proposals called "one-hop DHTs" which effectively go back to the O(n) state of consistent hashing, [7]. DHTs have had some real impacts and these include:

- BitTorrent uses DHTs to select trackers for their "trackerless" operation,
- Memorycach is a memory cach system used by Facebook, Live Journal, Wikipedia, etc to map objects only cache nodes.

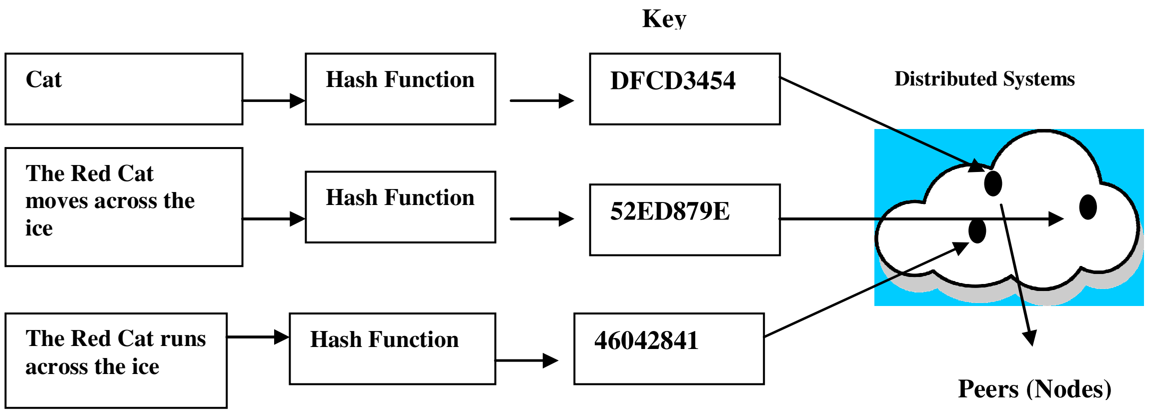

Fig. 2.2: Distributed Hash Table Process

In these DHTs, all nodes are assigned one or more random identifiers, called nodeids, typically from some space "keyspace" (e.g., all 160-bit numbers). In [19], "Keys" are also mapped into this same identifier/keyspace by computing some function F(k) that maps any input key k ({0,1}*) into the keyspace ({0,1}^160). One often uses a cryptographic hash function like SHA-1 for this F, as it has the property of *collision resistance*; namely, that it is very difficult to find two inputs k and k' for which F(k) = F(k') yet k != k'. (As a slight digression, if SHA-1 indeed behaves like a random function, then finding any such pair k,k' that "collide" will require searching roughly 2^80 numbers for 160-bit values, i.e., the "Birthday bound". There's been some very clever attacks against SHA-1 in the past few years [5], to bring this down to around 2^63, but that's perhaps not that relevant to our discussion, and one could always switch instead to a longer function, such as the 256-bit aptly-named SHA-256, and NIST currently has another competition going on to determine the next-generation cryptographic hash function.)

Anyway, we describe the basic approach of DHTs using ***Chord*** as an example, because we think it is the easiest to explain. We first start out with an identifier space that is comprised of all 160-bit numbers, i.e., [0, 2^160-1]. In ***Chord***, keys are the SHA-1 hash of URLs, and nodeids are the SHA-1 hash of nodes' IP addresses, [5]. We say that a node is a ***successor***(or ***root***) of a key if it is the first node that is clockwise from the key, as you go around the ring. DHTs poses how we can design a system where each node only knows some small fraction (typically O(log n)) of the system's nodes, [27]. In Chord, as in most other DHTs, these known nodes are drawn from a special distribution, one that allows any node to find a key's node successor in a short number of routing hops (typically O(log n)) when we use the system's nodes as a routing overlays. Thus nodes in a multi-hop DHT play the role of both *routing* lookups and *storing* keys/value pairs. **Chord** maintains two different types of "links" in its routing table, [27]: (1) *successor links* which map to its immediate *k* nodes in a clockwise direction around the ID ring (i.e., nodes with next-highest ids), where *k* successors are known instead of just a single one for fault-tolerance; (2) *fingers* which provide long-distance shortcuts for efficient routing. In [19], these *finger* links are drawn from an exponential distribution, so a node with id *me* knows the node successors for the ids {*me+2^1, me+2^2, me+2^3, … me+2^log n*}. In other words, it has a link to some node 1/2 around the ring from it, 1/4 around the ring, 1/8, 1/16, and so on.

Content addressable network (CAN) is a distributed Internet-scale DHT-based infrastructure that provides hash table-like functionalities, [31]. Different from the one-dimensional space in Chord, the DHT space in CAN is a d-dimensional Cartesian space. The *d*-dimensional space is further dynamically partitioned among all nodes and each node only maintains its own individual and distinct zone in the space. In CAN, every node maintains 2*d* neighbors, thus the node connectivity is 2*d*. Here the notion of "neighbors" means two zones that overlap along *d* - 1 dimension and have neighbors on one dimension. As a result, CAN dose not need to introduce complex structures such as long links (e.g., the finger table in Chord) connecting nodes, which are further away from each other in the d-dimensional space, in order to improve the connectivity and reduce the complexity of routing. Note that d is a constant independent from the number of nodes in the system, which means that the number of neighbors each node maintains is a constant, no matter how many nodes the CAN system may have.

2.3 Physical Models of Distributed Systems
Distributed systems are undergoing a period of significant change traceable to a number of significant trends as mention in [13]:

- the emergence of pervasive networking technology
- the emergence of ubiquitous computing together with the desire to support user mobility in distributed environment
- the increasing demand for multimedia services across distributed systems
- the view of distributed systems as a utility.

The physical model of distributed systems is a representation of the constituents underlying hardware components of a distributed system that hides the specific details of the computer and networking technologies employed, [10]. Some of the physical models include;

Baseline Physical Model: a distributed system is seen as one in which hardware or software components located at networked computers communicate and coordinate their actions and activities merely by passing messages. This model is a minimal physical model of a distributed system, in which an extensible set of computer nodes interconnected by a computer network for the required passing of messages. Beyond the baseline model, we carefully identify three generations of distributed systems.

Early Distributed Systems: these are systems that emerged in the 1970s through early 1980s in attempt to the response of emergence of local area networking (like a typical LAN) technology, such as Ethernet. These systems usually consisted of between 10 and 100 nodes interconnected by a LAN, with very limited Internet connectivity and supported a small variety of services like shared local printers and file servers, email and file transfer across the Internet. Personal systems were largely uniform and openness was not a major concern. Issue of quality of service was still very much a challenge and was a crucial point for much of the research around such early systems.

Internet-Scale Distributed Systems: enhancement on the Internet platform leads to larger-scale distributed systems which started to emerge around the 1990s in response to the dynamic growth of the Internet during this time (example, during the launch of Google search engine in 1996). In such systems, the underlying physical infrastructure consists of a physical model as explained in [19] that is, an extensible set of nodes interconnected by a network of networks (the Internet). Such systems exploit the infrastructure provided by the Internet to become global. They integrate large numbers of nodes and provide distributed system services for global organizations and across organizational platforms. The level of heterogeneity in such systems is significant in terms of networks, computer architecture, operating systems, languages employed and the development and implementation teams involved. This has led to an increasing focus on open standards and related middleware technologies such as CORBA and more recently, web services. Also, services were implemented to provide end-to-end quality of service properties in such global large scale systems.

Contemporary Distributed Systems: In previous distribute systems above, nodes were typically desktop computers and therefore relatively static (that is, remaining in one physical location for complete periods), discrete (not embedded within other physical entities) and autonomous (to a large extent independent of other computers in terms of their physical infrastructure). The basic trends identified in the above section have resulted in further developments in physical models.

Distributed Systems of Systems: According to [www.sei.cmu.edu], report discusses the emergence of ultra-large-scale (ULS) distributed systems. The report focused on complexity of modern distributed systems by referring to such (physical) architectures as systems of systems (mirroring the view of the Internet as a network of networks). A system of systems can be defined as a complex system consisting of a series of subsystems that are systems in their own right and that come together to perform a particular task or tasks. Example of a system of systems, consider an environmental management system for flood prediction. In such a scenario, there will be sensor networks deployed to monitor the state of different environmental parameters relating to rivers, flood plains, tidal effects and so on.

This can then be integrated with systems that are responsible for predicting the likelihood of floods, by running (often complex) simulations on, for example, cluster computers, as discus in [25]. Other systems may be established to maintain and analyze historical data or to provide prompt alerting systems to concern stakeholders via mobile phones. The requirements for sharing within distributed environment lead to a need for a different type of service – one that supports the persistent storage of data and programs of all types on behalf of clients' nodes and the consistent distribution of up-to date data.

3. SYSTEM DESIGN METHODOLOGY AND ARCHITECTURE

We identified the sharing of resources (storage space and data) as a major goal for distributed systems. Also, suitable system architecture is pivotal for developing and implementing distributed storage system for a distributed network environment.

In this work, we have adopted an abstract data structure type, the DHT together with an abstract file system model, the Sun Network File System (Sun NFS) for file service to act as a storage system services using the Hadoop system. We describe the Sun NFS in some detail, drawing on our simpler abstract model to clarify its architecture. The Andrew File System (AFS) is then described, providing a view of a distributed file system that takes a different approach to scalability and consistency maintenance.

However, for our distributed file storage system, we combined the Sun NFS and AFS to have a Unified File Service Architecture (UniFSA). UniFSA is an abstract architectural model that underpins and integrates Sun NFS and AFS which is based upon sharing of functionalities between three major modules of – client component that emulates an ordinary file system interface for application programs, and the server components, that perform operations for client on directories and on files.

Sun NFS: provides transparent access to distributed and remote storage with data for client programs running on UNIX and other systems. The client-server relationship is symmetrical, in which case each computer or node in an NFS network can act as both a client and a server (which assumes a P2P topology), and the files at every node can be made available for remote access by other node. Any node can be a server, sending some of its files, and a client, accessing files on or from other nodes. But it is ideal to configure larger installations with some nodes as dedicated servers and others as workstations.

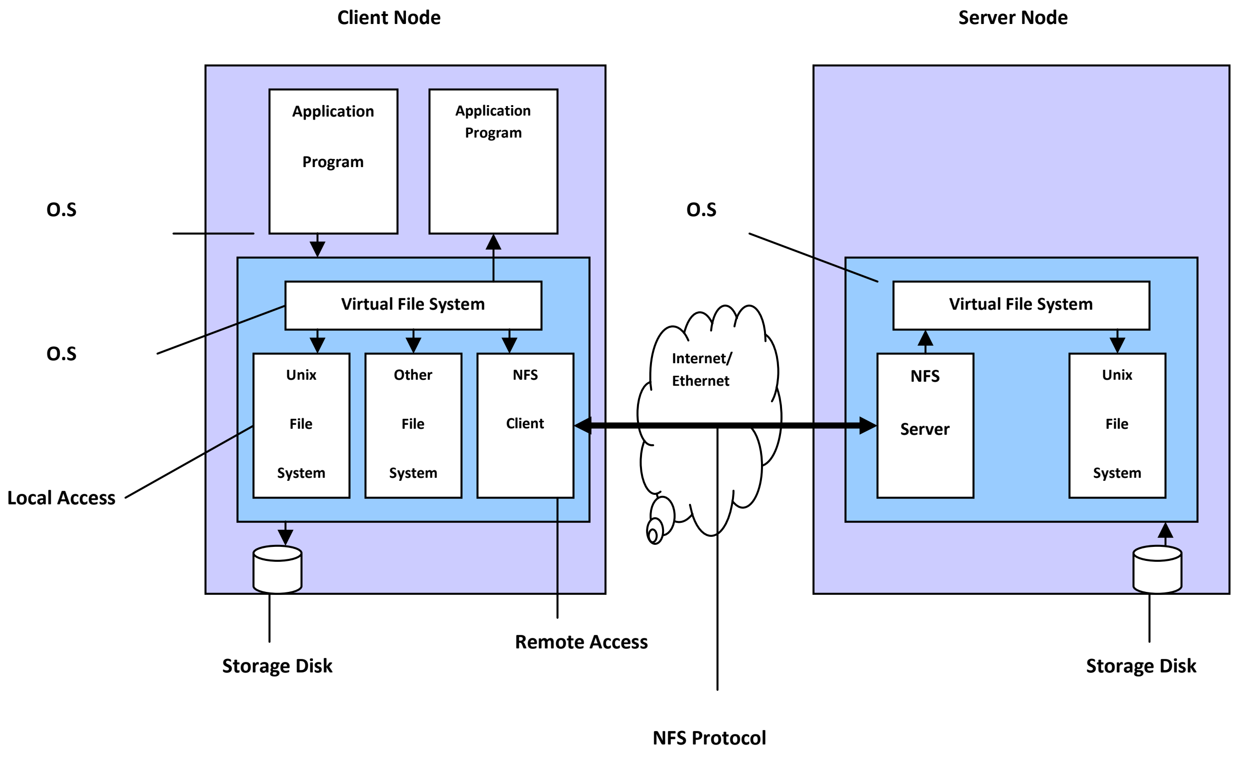

Figure 3: Sun NFS Architecture, [13]

In Fig. 3 above, the Sun NFS Architecture follows the abstract model mention explained the preceding sections. All implementations of the NFS in Fig. 3 support the NFS protocol, which is a set of remote procedure calls that provide means for the client node to perform either a read or write operations on a remote file store (typically on the server node). The NFS protocol is operating system independent, but was developed for use in a network environment. The Sun NFS server node module resides in the kernel on each computer that acts as the NFS server node at any instance. Requests refereeing to data or files in a remote file storage system, are translated by the client node module to NFS protocol operations and then passed to the NFS server node module at the computer holding the required data.

Both the NFS client and server nodes modules communicate using remote procedure calls of standard UDP or TCP. NFS in its design and implementation differs from ADS with the major difference mainly attributed to the identification of scalability. **AFS**: is designed to perform well with an increasing number of active nodes than other distributed file storage systems. The main strategy adopted for achieving scalability is the caching of whole files in client nodes.

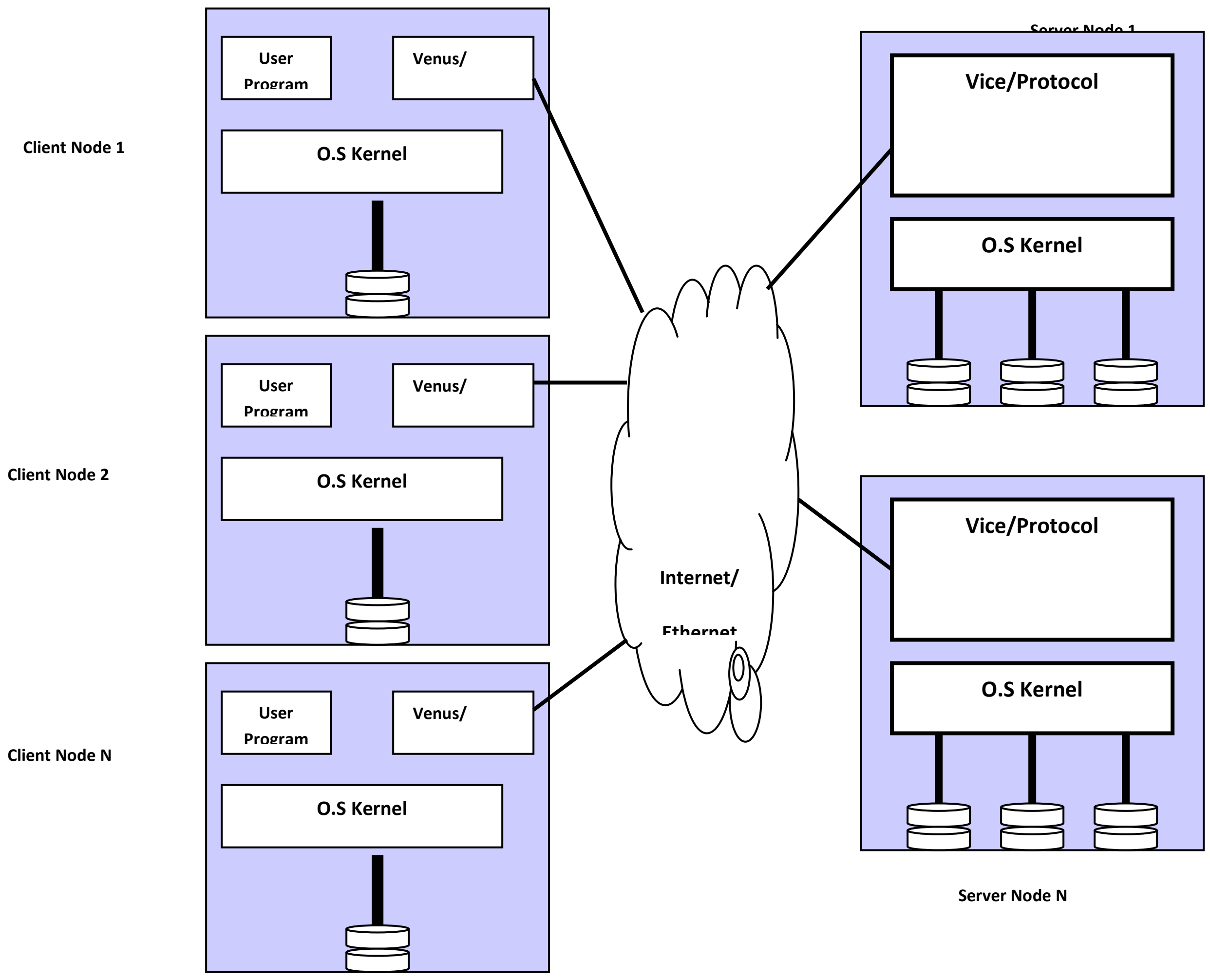

Figure 3.2: Architecture of Distributed Processes/Nodes in AFS

From the system architecture of distributed process and nodes in Fig. 3.2, AFS is implemented as two program components (Venus and Vice). Vice is the server node program that runs as a user-level O.S process in each server node, and Venus is a user-level process that runs in each client node and corresponds to client module in our abstract model of the DHT. The O.S kernel in each node (either client or server node) is a modified version of the system bootstrap system directories (BSD). The modifications are designed to intercept open, close and some other file system calls when they refer to files in the shared name space and them to the Venus process in the client node.

From the above systems architectures (for Sun NFS and AFS) in Figure 3.1 and 3.2 respectively, we exploit the functionalities of these systems by unifying their communication procedure of the global system state of available storage spaces on various nodes in the distributed environment by integrating the key modules of the architectures into Unified Distributed Storage System (uniDSS). The uniDSS architecture is illustrated bellow in Fig. 3.3;

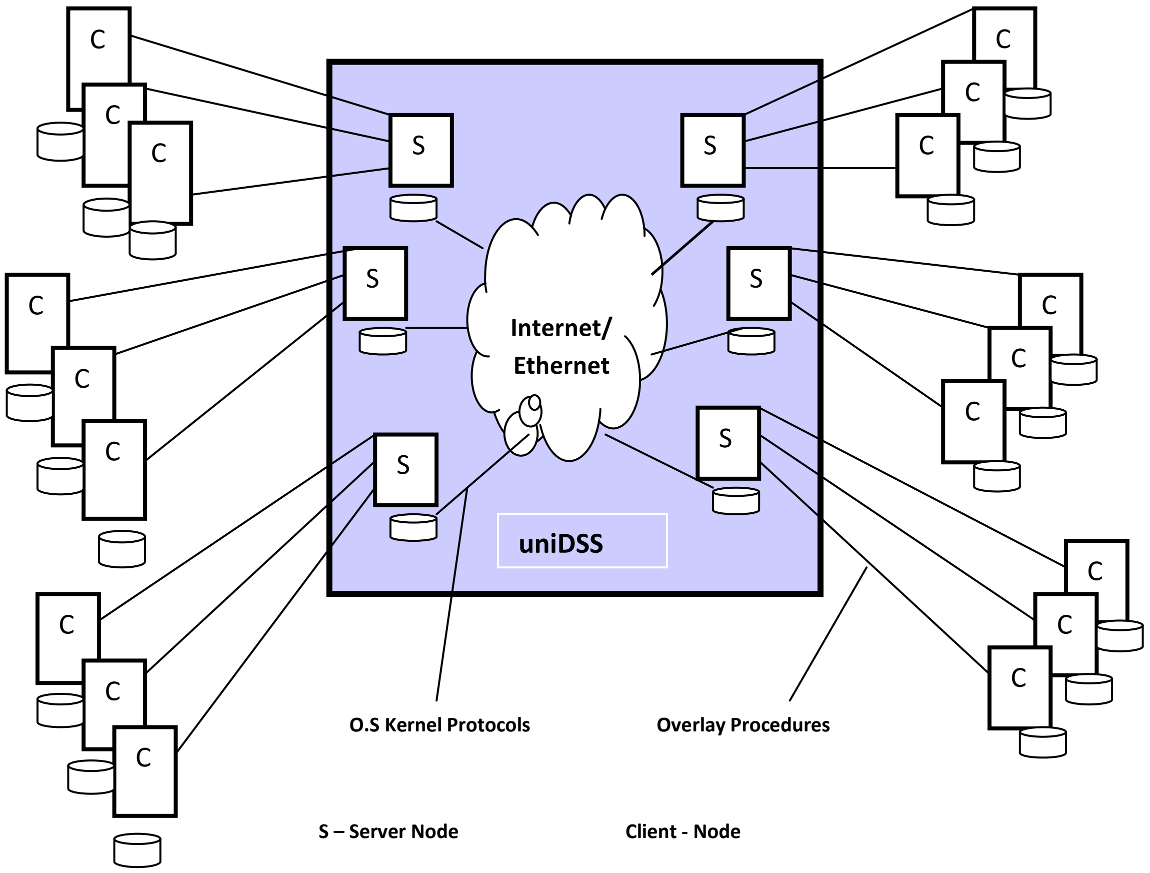

Figure: 3.3 Unified Distribute Storage System (UniDSS)

UniDSS is a self-managing storage layer designed and integrated to run on a cluster of workstations and to handle Internet or Ethernet service workloads. UniDSS has all of the service properties required for efficient *read* and *write* data operations: high throughput, high concurrency, availability, incrementally scalability, and strict consistency of its data. Behind these properties, the UniDSS platform hides all of the mechanisms used to access, partition, replicate, scale, and recover data. Because these complex mechanisms are hidden behind the simple UniDSS platform interface (the JDK), users' only need to worry about service-specific logic when implementing a new service. The difficult issues of managing persistent state are handled by the UniDSS platform.

3.1 Assumptions and the Design Principles for Implementation

We mention some of the design principles that guided us while building our distributed hash table UniDSS. We also state a number of key assumptions we made regarding our distributed environment, failure modes that the UniDSS can handle, and the workloads it will receive.

Separation of concerns: the clean separation of service code from storage management simplifies system architecture by decoupling the complexities of state management from those of service construction. Because persistent service state is kept in the UniDSS, service instances can crash (or be gracefully shut down) and restart without a complex recovery process. This greatly simplifies service construction, as users' need only worry about service-specific logic, and not the complexities of data partitioning, replication, and recovery.

Appeal to properties of clusters: in addition to the properties listed above we require that our cluster is physically secure and well-administered. Given all of these properties, a cluster represents a carefully controlled environment in which we have the greatest chance of being able to provide most (if not all) of the service properties.

Design for high throughput and high concurrency: given the workloads that may be available, the control structure used to effect concurrency is critical. Techniques often used by web servers, such as process-per-task or thread-per-task, do not scale to our needed degree of concurrency. Instead, we use asynchronous, event-driven style of control flow in our UniDSS, similar to that espoused by modern high performance servers [5, 20] such as the Harvest web cache [8] and Flash web server [28]. A convenient side-effect of this style is that layering is inexpensive and flexible, as layers can be constructed by chaining together event handlers. Such chaining also facilitates interposition: a ``middleman'' event handler can be easily and dynamically patched between two existing handlers. In addition, if a server experiences a burst of traffic, the burst is absorbed in event queues, providing *graceful degradation* by preserving the throughput of the server but temporarily increasing latency. By contrast, thread-per-task systems degrade in both throughput and latency if bursts are absorbed by additional threads.

3.2 Assumptions

We assume that if one node on the UniDSS cannot communicate with another, because the successor or other node has stopped executing (due to a planned shutdown or a crash); we assume that network partitions do not occur inside the distributed environment, and that UniDSS components are fail-stop. The need for no network partitions is addressed by the high redundancy of the network components (of several Server nodes), as previously shown in Figure 3.3. We have endeavoured to make fail-stop behavior in the UniDSS by having it terminate its own execution if it encounters an unexpected condition, rather than attempting to gracefully recover from such a condition. These strong assumptions are applicable in practice; example, it is rare to experience an unplanned network partition in a distributed platform. We further assume that systems failures in the distributed environment are independent. We replicate all durable data at more than one place in the distributed platform, but we assume that at least one replica is active (has not failed) at all times. We also assume some degree of synchrony, in that processes take a bounded amount of time to execute tasks, and that messages take a bounded amount of time to be delivered.

We make some assumptions also about the workload presented to our distributed hash tables. The table's key space is the set of 64-bit integers; we assume that the number of keys available over this space is even (i.e. the probability that a given key exists in the table is a function of the number of values in the table, but not of the particular key). We don't assume that all keys are accessed equally, but rather that; the *working set* of hot keys is larger than the number of nodes in our UniDSS. We then assume that a partitioning strategy that maps fractions of the *keyspace* to distributed nodes based on the nodes' relative processing speed will induce a balanced workload. Finally, we assume that tables (DHTs) are large and long lived. Hash table creations and destructions are relatively rare events: a common case is for hash tables to serve read, write, and remove operations.

3.3 UniDSS Distributed Hash Table Architectural Framework

In this section, we present the architectural framework for UniDSS implementation of a distributed hash table. Figure 3.4 illustrates our hash table's architecture, which consists of major components: client nodes, server nodes, UniDSS library and storage replicas.

Client Node: a client node component consists of service-specific processes running on a client machine that communicates across the network with one of many service instances running in the cluster. The mechanism by which the client node selects a service instance is not considered within the scope of this work, but it typically involves DNS round robin according to [7], a service-specific protocol, or level 4 or level 7 load-balancing switches on the edge of the distributed system. Example of a client node is a web browser, in which case the service and sever node would be a web server responding to a request.

Server Node: a Server node consists of cooperating O.S kernel procedure or processes, each of which we call a service instance. Service instances communicate with network clients nodes and perform some application-level function. Server nodes may have soft state (state which may be lost and recomputed if necessary), but they rely on the hash table to manage all persistent state.

UniDSS Library and Hash Table API: The UniDSS library is a Java class library that will present the hash table API to server node. The library accepts hash table operations, and cooperates with the server storage replicas to realize the table operations. The library contains only soft state, including metadata about the distributed system current configuration and the partitioning of data in the distributed hash tables across the storage replicas. The UniDSS library acts as the two-phase commit coordinator for state-changing operations on the distributed hash tables.

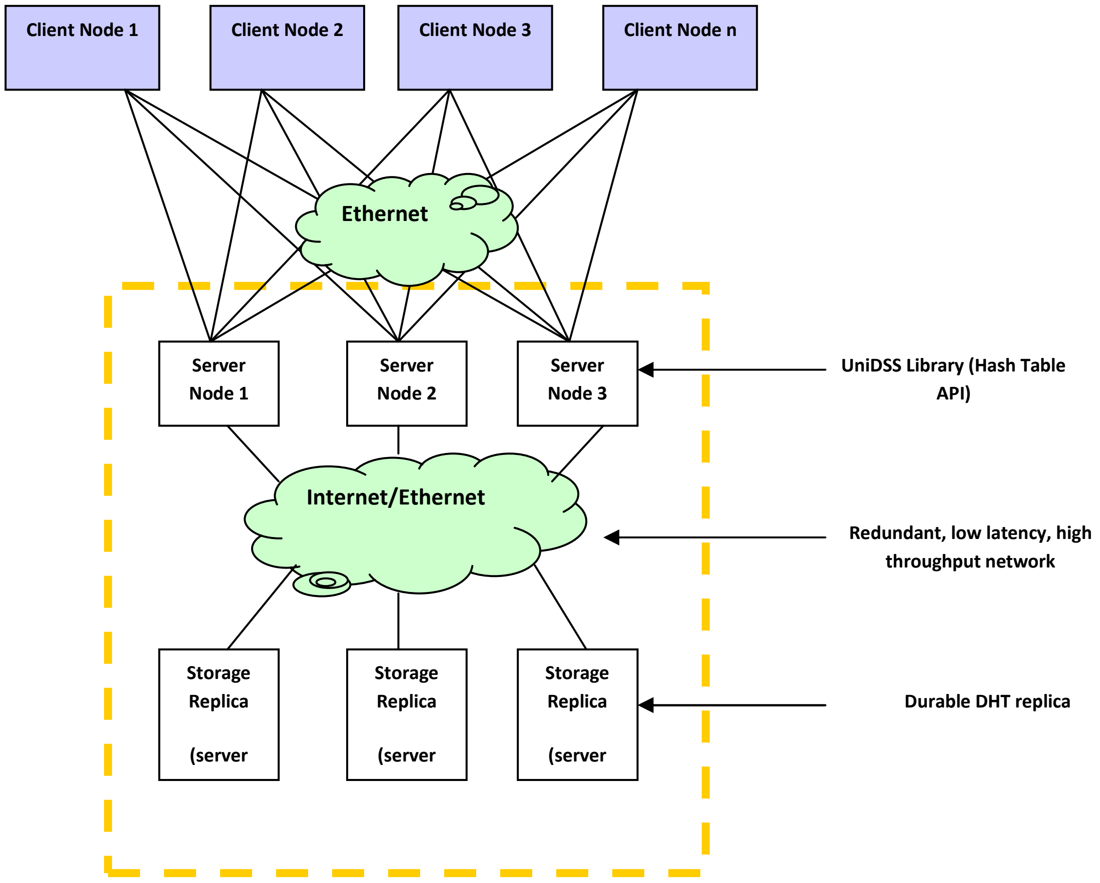

Figure 3.4: UniDSS DHT Framework

The hash table API is the boundary between a service instance and its UniDSS library. The API provides services with;

put(), get(), remove(), create(), and destroy() operations on hash tables.

Each operation is atomic, and all services see the same coherent image of all existing hash tables through this API. Hash table names are strings, hash table keys are 64 bit integers, and hash table values are opaque byte arrays; operations affect hash table values in their entirety.

Server Node Replica: Replica nodes are the system components that manage durable data. Each replica node will manage a set of network-accessible single node hash tables. A replica node consists of a buffer cache, a lock manager, a persistent chained hash table implementation, and network stubs and skeletons for remote communication. Typically, one replica node per CPU in the distributed system is suitable.

4. ALGORITHM AND IMPLEMENTATION REQUIREMENTS

The algorithm for the UniDSS protocol specifies how to find the storage locations of keys, how new storage nodes join the distributed system, and how to recover from the failure (or planned departure) of existing nodes. This section explains a simplified version of the UniDSS protocol that most distributed storage systems do not handle; like concurrent joins or failures.

4.1 The Algorithm for UniDSS Hash Table

In a DHT, the basic understanding is that hashing (hash function) assigns keys to nodes as follows: Identifiers are ordered in identifier circle modulo 2^m. Key *k* is assigned to the first node N_1 whose identifier is equal to or follows the identifier of *k* in the identifier space. This node is the *successor node* of key k, and it is denoted as *successor(k)*. If the identifiers are numbers from 0 to $2^m - 1$, the successor node *successor(k)* is the first node.

It follows that for any set of *N* nodes and *K* keys, then:

- Each node is responsible for at most $((1 + \varepsilon)K)/N$ keys
- When an $(N + 1)$st node joins or leaves the network, responsibility for $O(K/N)$ keys changes hands (only to or from the joining or leaving nodes on the distributed platform).

Table 1: Definition of variables for nodes and *m*-bit identifiers

Notation	**Definition**
identifier[k]. start	$(n + 2^{k-1}) \bmod 2^m$, $1 \le k \le m$
.interval	*(identifier[k].start, identifier[k + 1].start)*
.node	*First node ≥ n. identifier[k].start*
successor	*the next node on the identifier; identifier[1].node*
predecessor	*the previous node on the identifier*

```
START // identifier nodes present on the distributed network

            Define successor identifier[1].node
            Let n.join(n)
                If(n)
                Int identifier.table(n);
            Then; update.others();
            // we can move keys in (predecessor, n) from successor
            Else // n is the only node in the distributed network
                For i = 1 to m
                identifier[i].node = n;
                predecessor = n;

        // initialize the identifier table of local node
        // n' is an arbitrary node already in the distributed network
                n.init_identifier_table (n')
                        identifier[1].node = n'.find_successor(identifier[1].start)
            Look for predecessor node
                Predecessor = successor.predecessor
                Successor.predecessor = n;

                for i = 1 to m – 1
                    if (identifier [i + 1]. Start ε [n, identifier[i].node])
                    identifier[i + 1].node = identifier[1].node;
                else
                    identifier[I + 1].node = n'.find_successor(identifier[i+1].start);

            //update all nodes whose identifier tables should refer to n

                n.update_other()
                    for i = 1 to m
                // find last node p whose i^th identifier might be n
                    p = find_predecesor (n – 2^(i - 1));
                    p.update_identifier_table(n,i);

        // if s is i^th identifier of n, update n's identifier table with s
            n.update_identifier_table(s, i)
                if (s ε[ n, identifier[i].node])
                identifier[i].node = s;
                p = predecessor // get 1st node preceding n
                p.update_identifier_table(s, i);

                update identifier table and nodes


                return n;

            END;
```

Figure 4: Algorithm for computing the UniDSS DHT

4.2 The Java Implementation Platform

We found that Java is an adequate platform from which to build a scalable, high performance UniDSS. However, serious issues may arise with the Java language and runtime. Whenever the garbage collector became active, it had a serious impact on all other system activity, and unfortunately, current JVMs do not provide adequate interfaces to allow systems to control garbage collection behavior. The type safety and array bounds checking features of Java will vastly accelerate our UniDSS development process, and help us to write stable, clean code. However, these features got in the way of code efficiency, especially when dealing with multiple layers of a system each of which wraps some array of data with layer-specific metadata. Java lacks asynchronous I/O primitives, which necessitated the use of a thread pool at the lowest-layer of the system.

This is much more efficient than a thread-per-task system, as the number of threads in our system is equal to the number of outstanding I/O requests rather than the number of tasks.

5. RESULTS AND DISCUSSIONS

Our experience with the distributed hash table design – UniDSS has taught us many lessons about using it as a storage platform for scalable services. The hash table was a resounding success in simplifying the construction of interesting services, and these services inherited the scalability, availability, and data consistency of the hash table. As we scaled our UniDSS for distributed hash table functionality implementation, we noticed scaling bottlenecks that weren't associated with our UniDSS. At 128bits, we approached the point at which 100 Mb/s Ethernet switches would saturate; upgrading to 1 Gb/s switches throughout the cluster would delay this saturation. We also noticed that the combination of our JVM's user-level threads and the Linux kernel began to induced poor scaling behavior as each node in the distribution platform opened up a reliable TCP connection to all other nodes in the distribution network. The protocol call processes began to saturate due to a flood of signals from the client and server nodes O.S kernel to the user-level thread scheduler associated with TCP connections with data waiting to be read or write.

6. CONCLUSION

This work presents a new persistent data management system that enhances the ability of distributed systems to support Internet services. This self-managing storage system, called a Unified Distributed Storage System (UniDSS), fills in an important gap in current cluster and distributed platforms by providing a data storage platform specifically tuned for services' workloads and for the distributed environment. This research focused on the design and implementation considerations of a distributed hash table - UniDSS, theoretically demonstrating that it has many properties necessary for Internet services (incremental scaling of throughput and data capacity, fault tolerance and high availability, high concurrency, and consistency and durability of data). These properties are achieved by carefully designing and developing UniDSS algorithm that handles recovery techniques in the hash table implementation to exploit features of the distributed environments (such as a low-latency network with a lack of network partitions). The hash table UniDSS simplifies Internet service construction by decoupling service-specific logic from the complexities of persistent state management, and by allowing services to inherit the necessary service properties from the UniDSS rather than having to implement the properties themselves.

REFERENCE

[1] E. Amir, S. McCanne, and R. Katz. An Active Service Framework and its Application to Real-Time Multimedia Transcoding. In *Proceedings of ACM SIGCOMM '98*, pages 178-189, Oct 2008.

[2] T. E. Anderson, D. E. Culler, and D. Patterson. A Case for NOW (Networks of Workstations). *IEEE Micro*, 12(1):54-64, Feb 2015.

[3] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless Network File Systems. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, Dec 2005.

[4] D. Andresen, T. Yang, O. Egecioglu, O. H. Ibarra, and T. R. Smith. Scalability Issues for High Performance Digital Libraries on the World Wide Web. In *Proceedings of IEEE ADL '96*, Washington D.C., May 1996.

[5] G. Banga, J. C. Mogul, and P. Druschel. A Scalable and Explicit Event Delivery Mechanism for UNIX. In *Proceedings of the USENIX 1999 Annual Technical Conference*, Monterey, CA, Jun 2015.

[6] BEA Systems. BEA WebLogic Application Servers. http://www.bea.com/products/weblogic/. 27 May, 2017.

[7] T. Brisco. RFC 1764: DNS Support for Load Balancing, Apr 2012.

[8] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J.Worrell. A Hierarchical Internet Object Cache. In *Proceedings of the 1996 Usenix Annual Technical Conference*, Jan 1996.

[9] A.D. Birrell et al. Grapevine: An Exercise in Distributed Computing. *Communications of the ACM*, 25(4):3-23, Feb 2004.

[10] D. DeWitt et al. The Gamma Database Machine Project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), Mar 1990.

[11] G. A. Gibson et al. A Cost-Effective, High-Bandwidth Storage Architecture. In *ASPLOS-VIII*, San Jose, California, 2008.

[12] J. H. Howard et al. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1), Feb 2008.

[13] P. Ferreira et al. PerDiS: Design, Implementation, and Use of a PERsistent Distributed Store. In *Recent Advances in Distributed Systems*, volume 1752 of *Lecture Notes in Computer Science*, chapter 18, pages 427-452. Springer Verlag, Feb 2000.

[14] V. S. Pai et al. Locality-Aware Request Distribution in Cluster-Based Network Servers. In *ASPLOS-VIII*, San Jose, CA, Oct 1998.

[15] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster Based Scalable Network Services. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, St.-Malo, France, Oct 2007.

[16] I. Goldberg, S. D. Gribble, D. Wagner, and E. A. Brewer. The Ninja Jukebox. In *The 2nd USENIX Symposium on Internet Technologies and Systems*, Boulder, CO, Oct 1999.

[17] G. Graefe. Encapsulation of Parallelism in the Volcano Query Processing System. In *ACM SIGMOD Conference on the Management of Data*, Atlantic City, NJ, May 2000.

[18] J. Gray. The Transaction Concept: Virtues and Limitations. In *Proceedings of VLDB*, Cannes, France, September 2001.

[19] S. D. Gribble and E. A. Brewer. System Design Issues for Internet Middleware Services: Deductions from a Large Client Trace. In *Proceedings of the 1997 USENIX Symposium on Internet Technologies and Systems (USITS 97)*, Monterey, CA, Dec 1997.

[20] J. C. Hu, I. Pyarali, and D. C. Schmidt. Applying the Proactor Pattern to High Performance Web Servers. In *Proceedings of the 10th International Conference on Parallel and Distributed Computing and Systems*, Oct 2008.

[21] A. Iyengar, J. Challenger, D. Dias, and P. Dantzig. High-Performance Web Site Design Techniques. *IEEE Internet Computing*, 4(2), Mar 2000.

[22] J. S. Karlsson, W. Litwin, and T. Risch. LH*LH: A Scalable High Performance Data Structure for Switched Multicomputers. In *Proceedings of the 5th International Conference on Extending Database Technology*, pages 573-591, Avignon, France, Mar 2006.

[23] O. Krieger and M. Stumm. HFS: A Flexible File System for Large-Scale Multiprocessors. In *Proceedings of the 1993 DAGS/PC Symposium*, pages 6 14, Hanover, NH, Jun 1993.

[24] E. K. Lee and C. A. Thekkath. Petal: Distributed Virtual Disks. In *ASPLOS VII*, Cambridge, MA, 2006.

[25] B. G. Lindsay. A Retrospective of R*: A Distributed Database Management System. *Proceedings of the IEEE*, 75(5):668-673, May 1997.

[26] W. Litwin, M. Neimat, and D. A. Schneider. RP*: A Family of Order Preserving Scalable Distributed Data Structures. In *Proceedings of the Twentieth InternationalConference on Very Large Databases*, pages 342-353, Santiago, Chile, 2004.

[27] P. V. Mockapetris and K. J. Dunlap. Development of the Domain Name System. In *ACM SIGCOMM Computer Communication Review*, 1998.

[28] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An Efficient and Portable WebServer. In *Proceedings of the 2009 Annual Usenix Technical Conference*, Jun 2009.

[29] M. Rosenblum and J. K. Ousterhout. The Design and Implementation of a Log Structured File System. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, 1991.

[30] Y. Saito, B. Bershad, and H. Levy. Manageability, Availability and Performance in Porcupine: a Highly Scalable, Cluster-based Mail Service. In *Proceedings of the 17th Symposium on Operating System Principles*, Kiawah Island, SC, Dec 2009.

[31] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and Implementation of the Sun Network Filesystem. In *Proceedings of the USENIX 1985 Summer Conference*, El Cerrito, CA, Jun 2005.

[32] J. Song, E. Levy, A. Iyengar, and D. Dias. Design Alternatives for Scalable Web Server Accelerators. In *Proceedings of the 2000 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS-2000)*, Austin, TX, April, 2000.

[33] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: A Scalable Distributed File System. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, St.-Malo, France, Oct 2007.