CLASH: A Protocol for Internet-Scale Utility-Oriented Distributed Computing

Archan Misra, Paul Castro
IBM T.J. Watson Research Center
{archan, castrop}@us.ibm.com

Jinwon Lee
Korea Advanced Institute of Science and Technology
jcircle@nclab.kaist.ac.kr

Abstract

Distributed Hash Table (DHT) overlay networks offer an efficient and robust technique for wire-area data storage and queries. Workload from real applications that use DHT networks will likely exhibit significant skews that can result in bottlenecks and failures that limit the overall scalability of the DHT approach. In this paper we present the Content and Load-Aware Scalable Hashing (CLASH) protocol that can enhance the load distribution behavior of a DHT. CLASH relies on a variable-length identifier key scheme, where the length of any individual key is a function of load. CLASH uses variable-length keys to cluster content-related objects on single nodes to achieve processing efficiencies, and minimally disperse objects across multiple servers when hotspots occur. We demonstrate the performance benefits of CLASH through analysis and simulation.

1. Introduction

Emerging distributed computing applications, such as massively multiplayer games (MMP), corporate messaging systems, and enterprise telematics (e.g., vehicle fleet management), will rely on cooperative data processing and storage by a set of peer servers distributed over the Internet. These applications can employ thousands of servers to process millions of data streams that represent queries or updates to data shared by many clients.

To improve the scalability of such systems, we may distribute the workload across multiple servers using some partitioning criteria. Clearly, workload distribution must be done judiciously: uneven or unpredicted workload skews can cause performance bottlenecks and server failures (hotspots). In this paper, we focus on a decentralized, adaptive workload distribution middleware that distributes the workload of streaming data applications over a dynamically varying pool of wide-area servers. To scale, servers rely on decentralized control to organize into an overlay network that accepts continuous queries from clients over a large set of data; both the queries and data may experience frequent updates. We are pursuing an "on-demand" workload allocation strategy conforming to the "computing as a utility" model inspired by Grid Computing concepts. In this view, a distributed computing infrastructure acts as a shared resource for a wide class of applications that require large, but varying amounts of computing power. Accordingly, our infrastructure design is guided by the following principles:

On-demand allocation of resources – in our model, customers deploy their applications on a common, shared infrastructure built by a utility provider to reduce operational costs. Utility providers charge customers according to the resources consumed by their applications. Provisioning applications in a fixed manner (e.g. peak load) is not ideal from the perspective of the customer or the utility provider. A better strategy is to dynamically assign resources to an application as load fluctuates. This minimizes the cost to the customer by reducing the amount of over provisioning necessary to guarantee a quality of service (QoS) level. The utility provider also benefits by maximizing the pool of free resources that can be directed to other applications.

Transparent operation with no central state – applications need not be aware of the on-demand operation of the computing utility. Applications y should continue to function and receive the specified level of QoS (e.g., latency) independently of the number of servers currently allocated for its distributed operation. Clients that communicate with an application may be mapped to different servers as a result of application migration but this is achieved through seamless redirection. To scale to the Internet, servers should only manage a limited amount of state to run applications; no server is required to maintain the global server allocation state.

·*Content-sensitive data placement* – for data-centric applications, it is important to preserve semantically relevant groupings of data for processing efficiency and reduction of overhead (e.g. communication and state

transfer costs). For example, in systems for processing subscriptions over data streams such as NiagaraCQ [6], Xfilter [1], efficient query processing at a single server is based on the ability to create efficient indices over streams and queries with intersecting attribute values. Queries in these systems are typically expressed as predicates over a small set of attributes and it is possible to achieve fairly efficient query processing if data objects or streams with similar attribute values are clustered on a single physical node. Thus, our goal is to cluster groups of "similar" application-specific objects (whether data or queries) on as few servers (ideally 1, if the workload was low enough) as possible.

1.2 Distributed Hash Table Overview

Researchers have proposed Distributed Hash Table (DHT) protocols [14][16][17][18] to organize highly distributed and loosely coupled servers into an overlay network for the purpose of storing massive numbers of data objects. In the DHT environment, both the addition and removal of servers as well as the placement of data objects on servers in the network, take place in a purely decentralized and dynamic fashion.

To store an object, the DHT protocol assumes that all data objects have an identifier key that can be hashed to a hash key. Since all servers are peers, the data object can be presented to any server, and the overlay network will forward the object from server to server until the object arrives at the server that manages the hash key derived from the object's identifier key. DHT proposals differ principally in the precise forwarding mechanism employed to ensure that this process occurs in $O(log(S))$ time, where S is the number of servers in the overlay network. Object retrieval works analogously to storage: any client can lookup an object in $O(log(S))$ time by querying any server with the desired object's identifier key.

Because of their robust nature (most implementations employ replication for fault tolerance), potential to scale to the Internet, and bounded lookup times, DHTs are an attractive substrate on which to build a large-scale distributed system. However, from the perspective of the computing utility model, there are two critical shortcomings to all basic DHT protocols that must be addressed first:

1. Basic DHT proposals do not automatically adjust workload distribution to alleviate hotspots, which ultimately limits the overall scalability of the approach. Often, there exists an implicit assumption that the query workload is uniformly distributed over the objects stored in the system and the principle focus is on ensuring a uniform partitioning of a hash space among a set of peer nodes (e.g., the use of "virtual servers" in [17]).

2. DHT overlay networks inherently distribute objects across as many servers as possible. While this lowers the probability of concentrated hotspots, this approach: a) randomizes objects at very fine granularity, making it difficult to achieve "clustering" of related objects, b) ignores the communication costs associated with queries for semantically related objects that now have to be replicated across many nodes, and c) forces a low server utilization model for individual applications since the data they access is fragmented across many servers.

To alleviate these shortcomings, we present Content and Load-Aware Scalable Hashing (CLASH), a redirection layer that can be used in conjunction with most DHT implementation. CLASH addresses the need for content-sensitive clustering of objects and hotspot elimination for skewed workloads in basic DHT schemes.

The rest of the paper is organized as follows. Section 2 reviews previous work in the area of load balancing for hash-based distributed systems. In Section 3 we explain how CLASH creates key groups. Section 4 explains the basic working of CLASH's binary splitting algorithm. Section 5 presents the distributed CLASH protocol. Section 6 presents results from simulation studies of CLASH. Finally, Section 7 concludes with a discussion of our plans for future research.

2. Related Work

The base implementation of DHT protocols, such as Chord [17], CAN [14], Pastry [16] and Tapestry [18]), implicitly assume a uniform workload in the hash space; load balancing is accomplished by ensuring a uniform partitioning of the hash space among the server nodes.

[17] proposes the use of log(S) virtual servers per physical server node, where S is the total number of nodes, to significantly reduce the probability of non-uniform address allocation in the hash space. Alternatively, CAN alleviates the unfairness problem by considering the address spaces being managed by several neighboring nodes in determining the contiguous chunk allocated to a new server node. CFS [7] considers the existence of heterogeneous peers and allocates the number of virtual servers in proportion to the actual processing capacity of a physical server.

Some recent papers have considered the possibility of skews in the DHT workload. [13] reuses the notion of virtual servers and aims to balance loads by essentially moving virtual servers from an overloaded node H to an under-loaded physical node L (in essence transferring the responsibility for a chunk of the hash space from H to L). [5] requires a client node to hash each object key to d (>=2) distinct hash values by using multiple hash functions. From the set of servers handling one or more

of these *d* distinct hash values, the client selects the least-loaded server for storing the data object. Neither of these proposals address our aim of clustering objects with similar attribute values on a single server as long as feasible.

Variable depth hashing (e.g. scalable distributed data structures), like that employed by CLASH, is useful in databases [9] and has been used in non-DHT environments to relieve hotspots [12][11]. While CLASH shares the notion of a dynamically variable address space range with [12], CLASH operates in the identifier key space, leaving the base DHT protocol unchanged. Moreover, while CLASH increases the depth selectively only for "hot" objects, [12] changes the entire hash space and does not discuss how client nodes obtain the correct value of the hash depth *i* and the server set associated with the hash value *v*.

3. Encoding Semantics in Hash Keys

All the proposed DHT implementations essentially operate in a two-step fashion:

1. Each object is assumed to possess an *identifier key*. This key is first hashed to a hash value (*hash key*).

2. The object is then forwarded to the server currently managing that hash key.

The method used to create identifier keys defines the semantics used to store and retrieve objects in a DHT and several methods can be used [4][15][19]. In CLASH, identifier keys encode hierarchical clustering relationships about objects. As an example of this encoding, a geographic area can be encoded in a hierarchical *N*-bit identifier key adopting a quad-tree formulation [8]. A large rectangular area can be split into 4 sub-regions with each sub-region receiving a 2-bit label corresponding to the first 2 bits of the *N*-bit identifier key. These areas can again be split into 4 with the sub-regions receiving a 2-bit label, which is appended to the key of the enclosing parent region. This process can be repeated until an *N*-bit key is generated.

In general, hierarchical identifier keys encode parent-child relationships for groups of objects. When viewed as a tree, the entire *N*-bit key uniquely describes a leaf group (which may be an individual object); keys with common prefixes define clusters of related objects. We express groups using a prefix notation "*", meaning "don't care." For example, for an *N*-bit key the label "11*" means a group of all keys with the prefix "11", which represents the 2^{N-2} remaining combinations of possible keys. The label "111*" is contained in "11*" and identifies the more select group of 2^{N-3} keys with prefix "111."

CLASH uses key groups to dynamically control the placement of semantically related objects (defined by the identifier key) on the same physical servers. The heart of the CLASH is the ability, within a DHT framework, to dynamically assign groups of objects to servers where the size of the group is load dependent. CLASH takes a "hot" group from an overloaded server and splits it into smaller groups according to the encoding of the identifier keys. These smaller groups get dispersed across additional servers. For "cooler" groups, CLASH attempts to combine these into a larger aggregate group on a single physical server based on the identifier key. In the next section we describe how this is done in more detail.

4. CLASH Binary Splitting Algorithm

In a DHT protocol, some arbitrary function *KeyGen*() generates an *N*-bit identifier key *k*. A hash function *f*() maps the space *K* of all possible identifier keys to a hash-space *H*, such that *h*=*f*(k) where *h* is an *M*-bit hash key. Each server belonging to *S*, the set of all servers, is responsible for managing a portion of the total *M*-bit hash-space *H*. The DHT protocols operate by mapping *h* to a unique server *s*, using the distributed function s ← *Map*(h) where *Map*() is defined by the specific DHT protocol.

CLASH uses the *KeyGen*() function to encode key grouping semantics in an identifier key for each object. CLASH clusters identifier keys into key groups and uses DHT mechanisms to store these groups on servers in the network. CLASH imposes a many-to-one semantics such that semantically related objects (as defined by the hierarchical identifier keys) get mapped to the same key group.

CLASH imposes a *Shape*() function before the *Map*() function in DHTs to map identifier keys to their current key group. CLASH identifies key groups through a virtual key and a *key depth*. *Shape*() takes the *N*-bit identifier key *k* and a depth *d*, and generates an *N*-bit virtual key *k'* by taking k_d, first *d* bits of *k*, as the most significant bits of the virtual key and sets the remaining (*N-d*) bits to 0s. The virtual key *k'*, along with its depth *d*, identifies a key group containing all the *N*-bit key identifiers whose first *d* bits exactly match the bit pattern k_d. Accordingly, any virtual key created with *depth = d* effectively groups 2^{N-d} distinct keys (all possible variations in the residual *N-d* bits). For example, using our wildcard notation, the key group "0110*" includes the 7-bit key identifiers "0110101" and "0110111". The virtual key for this key group is "0110000" with depth = 4.

We can legitimately increase or decrease the depth of the virtual key to change the granularity of key grouping. Clearly, increasing *depth* from its current value *d* to *d*+1 increases the level of differentiation—the entire set of 2^{N-d} identifier keys that were identical in their initial *d* bits are now partitioned into two subsets of

cardinality 2^{N-d-1}, with the two subsets having an identical d-bit prefix, but differing in the $d+1$th bit. For example, when the set "0110*" with a depth of 4 is expanded, we effectively create two subsets of virtual key groups, "01101*" and "01100*", each with a depth of 5. On expanding these virtual key groups to their full N (=7) bit value, we can see that "01100*" equals the original "0110*" representation (the full expansion of which is "0110000" or decimal 48). On the other hand, the "01101*" virtual key expands to "0110100" (decimal 54). By applying a basic DHT *Map()* to our virtual keys, we can see that all objects in the sub-group "01100*" will map to the same hash value as the original larger group "0110*", and thus be directed towards the same DHT server. On the other hand, the sub-group "01101*" will map (most likely) to a different hash value, and could thus, be directed towards a server different from the one handling the original "0110*" group of objects. It should now be clear that *increasing the depth for a group by 1 from its current value halves (in the key identifier space) the set of identifier keys of that group that are being handled by the current server.*

For effective load balancing, CLASH uses *variable depth keys* to make the mapping of identifier keys to hash keys non-deterministic and load-aware. Key depth depends on the load a key group contributes to the overall load of a server. Different key groups have different depths. To make this clearer, we represent key group splitting as a logical binary tree. Figure 1 shows the logical binary tree created when we start with an initial virtual key value of "011*" (*depth* = 3), and iteratively increase the depth at the appropriate branches of the tree to distribute the load. For the purposes of clarity, the figure also shows the hypothetical identity of servers (obtained through the transformation *Map(f(k'))* that manage a specific sub-tree of the initial identifier space. When server s_0 (managing the virtual key group

Figure 1. Load-balancing using binary splitting

"011*") gets overloaded, it increases the depth, creating two new virtual key groups "0110*" and "0111*. While s_0 continues to manage objects with hash values "0110*", it off-loads the responsibility for the key group "0111*" to some other "child" peer-server (randomly chosen as s_{12} in Figure 1). Server s_{12} can subsequently split the key group "0111*" further, creating finer key groups "01110*" and "01111*". While s_{12} continues to assume responsibility for "01110*", it can request another "child" peer server (s_5 in Figure 1) to handle the key group "01111*". Subsequently, s_{12} can again split the key group "01110*", offloading the key group "011101*" to "child" server s_7.

While increasing the depth serves to split the workload between the parent node and a "child" peer server, an appropriate reduction in the depth provides a mechanism for performing greater clustering when the workload decreases. To avoid the complications that can arise from consolidation attempts at intermediate nodes (e.g., rolling back groups "0110*" and "0111*" to "011*" in Figure 1), our load-consolidation algorithm works in "bottom-up" fashion, with "leaf" nodes (those engaged in currently managing key groups at the leaves of the logical tree) informing their parents of the current workload. Under conditions of under-load ("cold" key groups), a parent node with cold left and right child leaf nodes can revert to the original depth (e.g., when key groups "011101*" and "011100*" (*depth*=6) are cumulatively "cold" enough, s_{12} can resume active management of the entire key group "01110*" (*depth*=5) and remove the child entries).

To store or query a data object, a client node must first determine the appropriate depth (and thus the virtual key) associated with the object's identifier key. Each of the leaf nodes in the logical tree (illustrated in Figure 1) corresponds to an "active" virtual key group (a key group currently being used to aggregate keys on the basis of a common prefix). For example, in Figure 1, s_0 is currently managing the key group "0110*" (*depth*=4), s_{12} is managing the key group "011100*" (*depth*=6), s_7 is managing the key group "011101*" (*depth*=6) and s_5 is managing the key group "01111*" (*depth*=5).

Clearly, each of the virtual key groups managed by any leaf node of the logical tree corresponds to a unique sequence of bits, since each sequence corresponds to a unique traversal of the tree from the root to the corresponding leaf. Since the bit sequence of any leaf virtual key group cannot be a prefix for the bit sequence of any other leaf virtual key group, there can be only one leaf virtual key group, *k'*, whose bits are an exact prefix match to the identifier key, *k*, of a specific object, and to which *k* currently belongs. The bit sequence "*k'* padded by *N-d* trailing zeroes" is then the current virtual key associated with *k*.
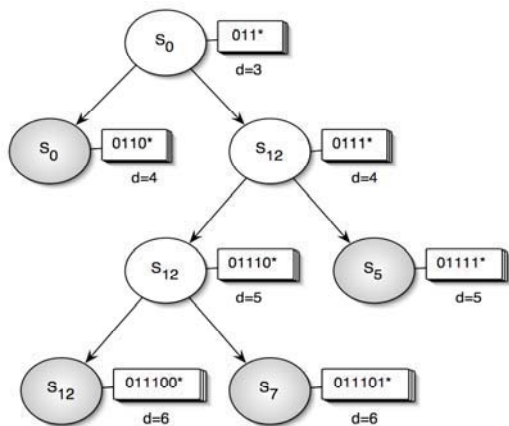
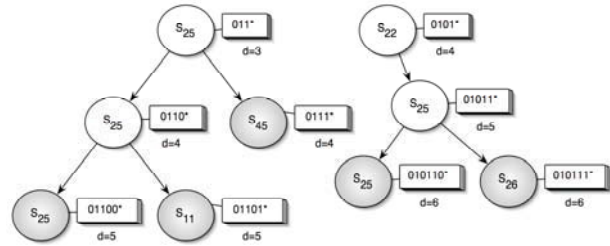| No.	Virtual Key Group	Depth	Parent ID	Right Child ID	Active
1	011*	3	-1	45	N
2	01011*	5	22	26	N
3	010110*	6	self	--	Y
4	0110*	4	self	11	N
5	01100*	5	self	-	Y

Figure 2. Key group information using Server Work Table

5. Server and Client Protocols

The CLASH protocol provides an efficient distributed solution to perform the prefix matching operation, when each server possesses knowledge of only the key groups that it is currently managing. Each server (currently responsible for one or more key groups) is assumed to monitor its own workload and determine the onset of overload when it must shed some load (e.g. this could be measured as latency, queue length, etc.). After identifying a key group (with *depth*=*d*) to be split (for example, the key group "01101*" with *d*=5), the server merely constructs two finer-grained key groups of depth *d*+1 that have the same prefix (the key groups "011010*" and "011011*" with depth 6). After zeroing out the remaining $N-(d+1)$ bits in the virtual key, the server uses the conventional DHT primitives to determine the right child server, which should handle the new right child key group ("011011*"). The server is guaranteed that the left child group will map back to itself ("01101*" and "011010*" expand to the same *N*-bit value). The server thus performs true load shedding: *keeping half the original key space for itself and transferring responsibility for the other half to another server*. As part of the load-shedding process, the server may also need to migrate state to the right child node, as necessary. A CLASH server does not need to explicitly determine a candidate child node, but simply relies on the underlying DHT protocol to determine the appropriate right-child node. The server sends an ACCEPT_KEYGROUP message to its right child node, transferring responsibility for the corresponding key group ("011011*" in our example). CLASH requires the "child" node to accept all ACCEPT_KEYGROUP messages, thereby allowing an overloaded node to always shed load to a peer. While the child node itself may be currently overloaded, it can then always choose to sub-divide the accepted key group further and shed its own load. Although the use of DHT implies that a CLASH server has a very high probability of picking a different node as its right child node, there is always the chance that the right-child node maps back to itself (e.g., if "01101*" and "01101*" both map to the same hash value) In such a case, the server can then simply increase the depth for this right key group again, thereby making another randomized attempt to select a different server node.

In CLASH, peer servers manage the information about the binary splitting tree in a distributed manner. Conceptually, each peer server maintains only the key groups assigned to it by *Map*(). This information is stored in a table data structure *ServerTable*, where the information for each entry consists of several fields, the most important of which are explained using the sample table in Figure 2, which shows the current server table (and the equivalent binary tree) for a hypothetical server (say server s_{25}). While the VirtualKeyGroup and depth fields are self-explanatory, the ParentID field stores the ID of the server that is managing the "parent" key group (for example, the parent for the key group "01011*" is server s_{22}, imply that s_{22} was responsible for managing "0101*" before being split). The ParentID field is -1 if this node is the "root" for this key group: root entries are an optional administrative tool to prevent servers from collapsing the workload beyond a minimum specified value (e.g., s_{25} is the root for the key group "011*"). The RightChildID field indicates the ID of the server that is handling the "right child" key group after a split. For example, entry 2 indicates that, on splitting the key group "01011*", s_{25} asked server s_{26} to manage the load for the right-child key group "010111*". Finally, Active is a Boolean-valued field that is "Y" if the entry is currently a leaf node in the logical tree. Thus, s_{25} is actively managing the key group "01100*", while the parent entries "0110*" and "011*" are currently inactive (i.e., are currently managed as smaller sub-groups)

To insert an object (data or query) with an identifier key *k*, a CLASH client node first "estimates" (e.g., picks at random) a depth *d*. After constructing the virtual key having the first *d* bits of *k* (and the rest zeroed out), it sends an ACCEPT_OBJECT message to the corresponding server (using conventional DHT to determine this server ID). The server's must respond to three possible cases (depending on whether the client's estimated depth *d* is correct or not), which we explain by using the sample server table in Figure 2:

a) The client had the right depth: (For example, if the client sent the (7-bit) key "0110001" with $d=5$). In this case, the server sends back an OK message.

b) Client had the wrong depth, but the server should be storing this object: (For example, if the client sent the 7-bit key "0110001" with $d=7$. Due to the random $Map()$ function, it is possible that $Map(f(011001))$ coincidentally turned out to be s_{25} as well). In this case, the server sends back an OK message, but with the corrected depth (5 in our example).

c) Client had wrong depth, and server is not currently responsible for this object. (For example, if the client sent "0101010" and $d=6$). In this case, the server sends back an INCORRECT_DEPTH message, specifying the longest possible prefix match between k and the current server entries. (In our example, the server sends back 4 to the client). We discuss the client's response to an INCORRECT_DEPTH next.

Unlike basic DHT, a CLASH client node wishing to insert an object (data or query) in the distributed system must first determine the correct depth, d_c, for its identifier key, before it can use DHT to identify the correct server. CLASH sacrifices some efficiency in object lookups to increase the availability of hot key groups. CLASH uses a modified binary-search technique to determine the correct depth d_c. CLASH clients can, on average, perform this lookup faster than the worst-case $O(log(N))$ bound of basic binary search. In essence, the client starts the binary search over the range $(0,N)$ by setting low=0 and high=N. If a client gets an INCORRECT_DEPTH message (indicating d_{min}) in response to an ACCEPT_OBJECT message sent with an "estimated" depth d, then it can adjust the low and high values based on the following properties:

1. If $d_{min} > d$, then the $d_c > d_{min}+1$. This message contains no further information about any upper bound on the correct depth d_c.

2. If however $d_{min} < d$, then d_c is both lower and upper as $d_c > d_{min}+1$, and $d_c < d$.

Convergence is guaranteed in CLASH. Simulation studies (to be described in the next section) show that, in practice, clients usually converge to the true depth much faster than $log(N)$.

Note that, in general, this estimation of the correct depth can be performed in O(log(N)) individual lookups by a server that uses this algorithm to query its peer servers, rather than assigning the lookup burden to the client.

6. Simulation and Performance

We have implemented a C++-based simulation engine for understanding the performance of CLASH. The simulator code extends the basic CHORD DHT simulation code available from [20]. We simulate a
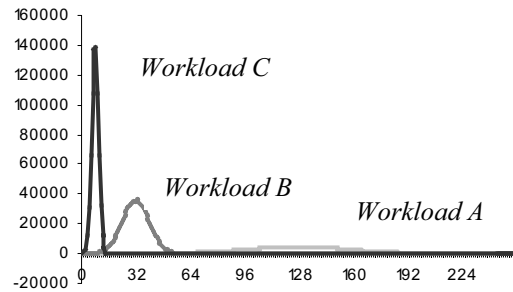

Figure 3. Workloads used in simulation

pseudo-distributed system for supporting long-lived queries over streaming data [2][3]. An example of such a system would be a distributed implementation of a telematics server such as Mobiscope [8], which supports continuous queries over moving objects. In the simulation, each server periodically computes a load value, based on the number of queries it currently stores and the cumulative data rate it currently handles. For query-processing applications, this load is usually linear in the data rate, and logarithmic in the number of queries. Overload and underload conditions are detected by comparing this load value to pre-defined thresholds.

We model an environment where queries are long-lived and stateless (no querying over historical data). The splitting process requires CLASH to appropriately migrate only query objects to a child; data packets are discarded after processing and are never migrated. In practice, the state to be migrated would be application-dependent (e.g., a distributed file system would migrate stored files), and the amount of migrated state should be counted as part of CLASH's distributed communication overhead. The logic employed by a server to select a key group to shed during overload, or a key group to consolidate during underload, is also outside the core specification of CLASH. In our implementation, we selected the "hottest" key group (the one with the highest load in the last measurement interval) for splitting during overload, and the "coldest" active key-group for possible consolidation during underload.

We consider the sources to be streaming data at a constant rate, where the key associated with each data packet represents a hierarchical encoding of some source (client) attributes. A source changes its key periodically, to reflect potentially dynamic changes in its attribute values. For example, in applications such as Mobiscope or multi-player games, the key represents the source location (in a real or virtual grid). This key remains unchanged as long as the client remains in the same (finest) grid. Accordingly, each data stream is associated with a virtual stream length (Ld), such that the key changes every Ld packets. Each client always has to perform a new lookup after every Ld packets

(since the key changes); once the client identifies the appropriate server for a virtual stream, it simply caches this server value and sends all subsequent packets with the same key to this server without incurring the overhead of DHT-based lookup. Of course, the client may have to perform additional lookups during a single virtual stream, if the corresponding server splits or merges and the client needs to be redirected. Varying Ld allows us to simulate various operating scenarios, with $Ld=1$ representing the extreme request-response where each data object has a different key. Skews in our system occur when either many data sources, or many query clients pick "similar" keys.

6.1 CLASH Simulation Parameters

We used several simulation studies to evaluate the performance of CLASH and its effectiveness in alleviating workload skews. To study the effect of skewed workloads over different ranges of the key space, we divide the $N=24$ bit key into 2 distinct portions: a base portion of X bits, and a remainder portion of 24-X bits. We present results with three different workloads, each with different amounts of skew in the distribution of the base $X=8$ bits. (The remaining bits are generated according to a uniform distribution). The three different workloads (A, B and C) are shown in Figure 3. Both data and query clients choose keys with the same skew. Clearly, workload A has the smallest skew (it's almost uniform) while workload C has the highest skew. Additionally, we also varied the intensity of the data streams: while data sources in workload A generated data packets at the rate of 1/sec, both workloads B and C correspond to source streams with data rates of 2 packets/sec.

To compare CLASH with a non-adaptive version of DHT, we also simulated the base Chord protocol, where the hash space is 24 bits and the length of the identifier key N is always fixed. We experimented with fixed identifier key lengths of $N=(2, 6, 12, 20)$; in our graphs, a plot for DHT(x) refers to a Chord experiment with fixed identifier key length equal to x. Simulation experiments are run for a total of 6 hours with 1000 servers, 100,000 client nodes, with servers checking their loads for potential overload or underload conditions every LOAD_CHECK_PERIOD (5 minutes).

We arbitrarily set the maximum acceptable load on a server to 90%, and the minimum (underflow) load to 54%, of the server's capacity. To capture the effect of varying workload skews, sources and query clients generated keys according to workload A for the first 2 hours, followed by workloads B and C over the next successive 2 hour intervals. The virtual stream lengths for each source client was generated according to an exponential distribution, with a mean of $Ld=1000$ packets. Each query client had an exponentially distributed lifetime of $Lq=30$ mins.

6.2 Utility-Style Load-Aware Distribution

Figure 4 shows the most important metrics used to evaluate CLASH performance vis-à-vis basic DHT. The
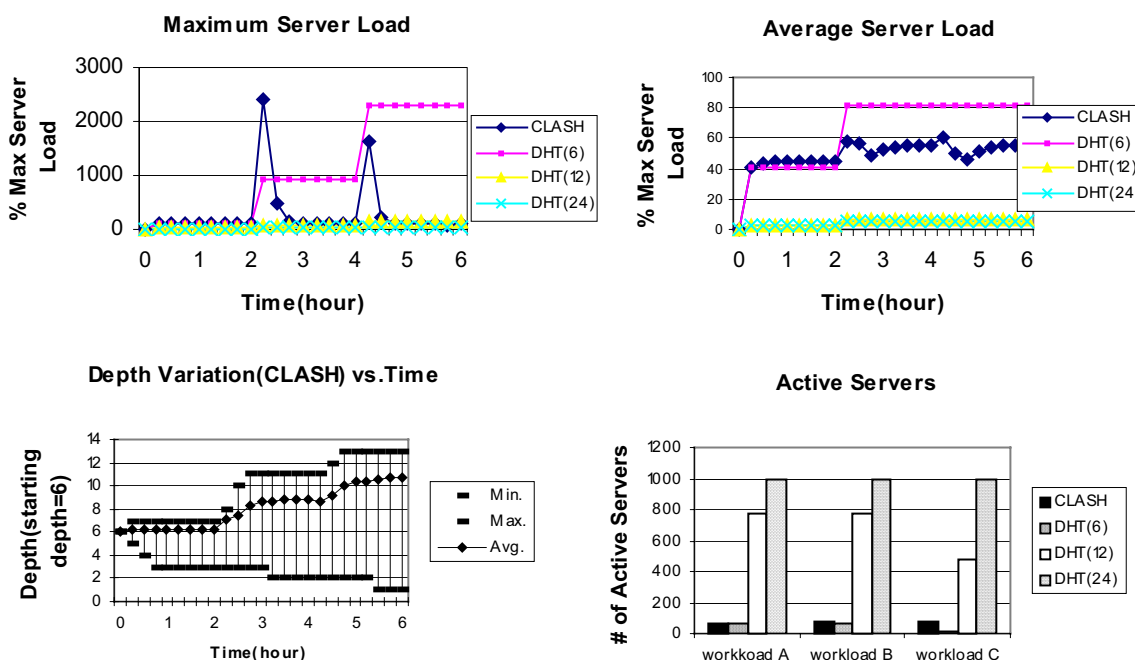

Figure 4. Server Load, Utilization, and Depth Variation for CLASH vs. Base DHT

figure shows the variation in average load of servers under CLASH and basic DHT (with N=6,12,24) over the size hour period (with 2 hours each of workload A, B and C respectively), as well as the maximum load experienced by any participating server. The figure also shows the number of servers actually used by CLASH and the basic DHT, as well as how the depth of CLASH's load-dependent tree evolves with changes in the workload.

These charts clearly explain the performance benefits of CLASH. From the average workload graph, we see that CLASH adjusts the depth to maintain uniformly high server load levels (~50-60% of an individual server's capacity) over the entire duration (across all workloads). On the other hand, using a very fine granularity in DHT (a large N, such as 12 or 24) leads to very low server loads. The graph shows that using a relatively small value of N (=6) causes high, but acceptable, average workloads. However, the real disadvantage of DHT emerges when we study the graph showing the maximum load level on any server. While CLASH is able to deal with skews and keep the maximum load on any server (after a small transient period, which would be absent in smoothly varying workloads) to less than 90% of capacity for all workloads, DHT with N=6 is unable to intelligently re-distribute the workload and cause the maximum load on a server to be as high as 25 times server capacity. Thus, for skewed workloads, DHT has to choose either between very low server utilization or unacceptably high localized workloads. The real problem is that DHT applies a uniform depth to the entire key space, unlike CLASH, which issues longer length keys only for portions of the key space that are currently "hot".

Figure 4 also shows the "on-demand" nature of server allocation in CLASH. The basic DHT protocol is non-adaptive—it uses all the servers to which any of the 2^N distinct identifier keys map. In contrast, CLASH attempts to use additional servers, only when specific key groups become unacceptably "hot" for a single server. Accordingly, Figure 4 shows that CLASH uses only a small fraction (approximately 70-80) of the 1000 servers, for all three workloads A, B and C, by essentially redistributing the allocation of the key space across servers. Basic DHT, on the other hand, either uses too many servers (for N=12, basic DHT ends up using ~450-800 servers based on the workload) with very low utilization, or uses a very small number of servers in an inefficient manner, leading to service outages at overloaded servers.

Figure 4 also shows how the average, maximum and minimum depth of the CLASH tree changes with changes in the workload. In general, when the total traffic intensity changes (workloads B and C have double the traffic rate of workload A), the average tree depth increases to encompass a larger number of servers. Moreover, the binary tree becomes progressively more unbalanced as the workload skew increases, since only a small fraction of the groups are now very hot and must be split to greater depth. Accordingly, we see that the variance between the minimum and maximum depth for the leaf nodes in the CLASH tree increases with an increase in the workload skew.

6.3 Additional Signaling Load under CLASH

Due to its distributed operation, CLASH introduces two forms of additional signaling load over basic DHT:

a) Servers must periodically exchange load information, as well as messages for merging and splitting groups.

b) Clients must determine the appropriate depth associated with their data or query key. Unlike basic DHT (where a lookup takes $O(log(S))$ message exchanges), a CLASH lookup requires $O(log(N)log(S))$ exchanges, since each of the $O(log(N))$ "guesses" for the right depth incurs $O(log(S))$ DHT overhead to determine the appropriate server (even if the iterative depth determination is performed by the first contacted server rather than the client). This depth determination occurs at the beginning of every virtual stream, and also whenever load-splitting/merging causes the client to be redirected to another server.

An adaptive mode of distributed operation also incurs additional state-transfer overhead during the splitting/ merging process—e.g., in a persistent-query application, a group splitting requires the parent server to migrate a subset of its stored queries to the target child. Figure 5 shows the total CLASH overhead for two different cases: (A) when the system has no query clients, and thus no state transfer overhead (since data objects are never stored) and, (B) when the system has 50,000 query clients. We simulated for both Ld=50 and 1000.

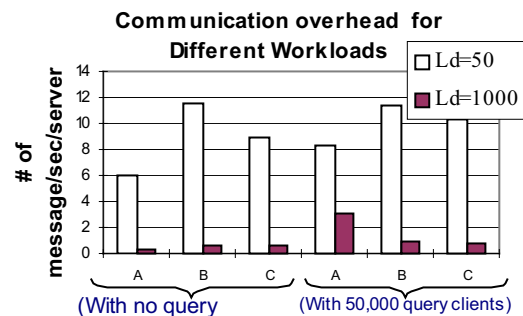Case (A), which captures the traffic overheads

Figure 5. Communication overhead with CLASH

ignoring any state transfers, shows that each CLASH server processes at most ~10-12 messages/sec across different workload skews and for different virtual stream lengths. Clearly, CLASH overheads are lower for longer-length streams (larger values of Ld), where keys change less frequently. Figure 5 also shows that, for our persistent query application, the state-distribution overhead (case B) adds very little overhead (~1-2 messages/sec/server) to the communication cost, even with 50,000 query clients. Clearly, if the amount of shared state is too large, then this transfer overhead becomes a constraint. Our on-demand model of server allocation thus appears to be better suited to emerging data-intensive applications (e.g., online games, streaming data filtering, vehicle tracking), rather than traditional distributed databases (e.g., distributed file systems). In such newer applications, server overload occurs primarily while processing transient data, rather than storing persistent state.

7. Conclusions and Future Work

We have described CLASH, a load-aware adaptive clustering protocol that can be combined with basic DHT primitives to provide a *utility-oriented middleware for distributed data-intensive applications*. CLASH works by dynamically varying the relevant portion of an object's key, such that keys with a common prefix are redirected to a common server. A CLASH server maintains only local state, namely only key groups that it currently manages. CLASH clients can determine the appropriate server for any N-length key in $O(log(N))$ individual key lookups by using a distributed algorithm that does not suffers from load concentration effects. In a utility environment, intelligent workload allocation by CLASH can reduce the number of physical servers utilized by as much as 80%, compared to basic DHT. We are currently building a CLASH-based middleware for online games, including an API that game servers use to indicate application overload and to distribute application-specific state. We are also working on supporting range queries in CLASH. For range queries, the CLASH overhead vis-à-vis DHT will decrease, since CLASH will cluster ranges of objects on a common server and thus incur lower query replication overhead.

8. References

[1] M. Altinel, M. Franklin, Efficient Filtering of XML Documents for Selective Dissemination of Information, VLDB 2000.

[2] M. Aguilera, R. Strom, D. Sturman, M. Astley, T. Chandra, Matching Events in a Content-based Subscription System, Eighteenth ACM Symposium on Principles of Distributed Computing (PODC 99),1999.

[3] S. Babu and J. Widom, Continuous Queries over Data Streams, SIGMOD Record 30(3), 2001

[4] M. Balazinska, H. Balakrishnan, and D. Karger, INS/Twine: A Scalable Peer-to-Peer Architecture for Intentional Resource Discovery, Pervasive 2002 - International Conference on Pervasive Computing, Zurich, Switzerland, August 2002

[5] J. Byers, J. Considine and M. Mitzenmacher, Simple Load Balancing for Distributed Hash Tables, 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03), February 2003.

[6] J. Chen, D. DeWitt, F. Tian, Y. Wang, NiagaraCQ: A Scalable Continuous Query System for Internet Databases, in ACM SIGMOD 2000, May 2000.

[7] F. Dabek, F. Kashoek, D. Karger, R. Morris and I. Stoica, Wide-Area Cooperative Storage with CFS, Proceedings of ACM Symposium on Operating System Principles (SOSP), October 2001.

[8] M. Denny, M. Franklin, P. Castro and A. Purakayastha, Mobiscope: A Spatial Discovery Service for Mobile Network Resources, 4th International Conference on Mobile Data Management (MDM), January 2003.

[9] R.J. Enbody, H.C. Du, Dynamic Hashing Schemes, ACM Computing Surveys, vol. 20, No. 2, June 1998.

[10] K. Hildrum, J. Kubiatowicz, J. Rao and B. Zhao, Distributed Object Location in a Dynamic Network, Proceedings of 14th ACM Symposium on Parallel Algorithms and Architectures (SPAA), August 2002.

[11] D. Karger , A. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidina, K. Iwamoto, B; Kim, L. Matkins, and Y. Yerushalmi. Web Caching with Consistent Hashing, In Proceedings of the 8th International World Wide Web Conference, Toronto, Canada, May 1999.

[12] R. Prakash and M. Singhal, Dynamic Hashing+ Quorum= Efficient Location Management for Mobile Computing Applications, Proceedings of ACM Symposium on Principles of Distributed Computing (PODC), August 1997.

[13] A. Rao, K. Lakshminarayanan, S. Surana, R. Karp and I. Stoica, Load Balancing in Structured P2P Systems, 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03), February 2003.

[14] S. Ratnasawmy, P. Francis, M. Handley, R. Karp and S. Shenker, A Scalable Content-Addressable Network, Proceedings of ACM SIGCOMM 2001.

[15] P. Reynolds, A. Vahdat, Efficient Peer-to-Peer Keyword Searching, Proceedings of Middleware 2003

[16] A. Rowstron and P. Druschel, Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems, Proceedings of IFIP/ACM International Conference on Distributed Systems Platforms (Middleware), November, 2001.

[17] I. Stoica, R. Morris, D. Karger, F. Kashoek and H. Balakrishnan, Chord: A Scalable Lookup Service for Internet Applications, Proceedings of ACM SIGCOMM, 2001.

[18] Ben Y. Zhao, John Kubiatowicz and Anthony Joseph, Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing,UCB Tech. ReportUCB/CSD-01-1141.

[19] F. Zhou, L. Zhuang, B. Zhao, L. Huang, A. Joeseph, J. Kubiatowicz, Approximate Object Location and Span Filtering on Peer-to-Peer Systems, Proceedings of Middleware 2003

[20] http://www.pdos.lcs.mit.edu/chord