Scalable Object Storage with Resource Reservations and Dynamic Load Balancing

Alex Aizman and Caitlin Bestler, Nexenta Systems, Inc. Santa Clara, CA 95050

***Abstract*—Scale-out distributed storage clusters require efficient selection of the storage targets holding replicas. This requires coordinated allocation of target IOPS, target persistent storage capacity and network bandwidth. The current generation of distributed storage solutions mostly select storage targets using consistent hashing. Implementations have become significantly more sophisticated than the algorithm originally presented for Amazon Dynamo. They (consistently) yield consistent locations for a given combination of content and clustered topology. But none of them consider dynamic factors, such as current work load or remaining capacity. Moreover, non-dynamic target selection leads to uneven distribution of network traffic. Uneven resource distribution underutilizes available capacity. We present an alternative solution, the Replicast protocol, which combines dynamic load balanced scheduling, multicast messaging and reservation-based payload delivery. Replicast has been implemented and is being deployed in a commercial solution called NexentaEdge. To evaluate Replicast for large and super-large clusters, we use a discrete event simulation framework called SURGE. In this paper, we compare the results of simulated object put benchmarks with models for conventional storage clusters that use consistent hashing and reliable unicast connections to transfer content.**

I. INTRODUCTION

Most current generation storage clusters use a variant of consistent hashing to distribute content. These algorithms deliver much better scale-out and, as the name implies, are based on a hash of the object or file name and/or object or file content. These solutions include Ceph, OpenStack Swift, GlusterFS and many others. These solutions all assume that scale-out storage requires a method for target selection which does not utilize location tracking metadata. As a storage cluster grows, maintaining explicit location tracking metadata becomes increasingly difficult. In a large cluster, adding or dropping storage targets becomes routine. A hash of the name or content determines where the data must be stored and hence where it can be located.

This paper describes a better method for selecting targets. The problem we want to address is efficient resource allocation in large distributed storage clusters. From the general resource allocation perspective, combined clustered resources should be shared, common and fungible. Users do not care which disks store which replicas of their data. But consistent hashing selects a fixed set of targets independently of their current utilizations. Scheduling become restricted and therefore sub-optimal. We argue that resource allocation decisions should be bounded only by the available resources. In a distributed storage cluster, allocations of network bandwidth, target IOPS and target storage capacity are all interdependent: network bandwidth gets allocated along a certain path, that path then must lead to the servers that have available IOPS and disk capacity to execute the corresponding storage transaction. We can look at distributed storage cluster as a special and specialized case of resource allocation and scheduling – see [1] for instance, where the problem of allocating network bandwidth is considered alongside tasks of machine scheduling and page caching.

We also believe that, in the context of resource management, allocating/scheduling network bandwidth separately from target IOPS is counter-productive. Historically networking and storage are largely done separately but subdividing the problem while honoring common layering principles may not optimize the ultimate answer – allocating just network bandwidth or, separately, just storage IOPS, may not provide the optimal resources to complete the entire task.

In this paper, we describe one networking protocol designed for storage clusters. We benchmark, compare and analyze its performance. Beyond the specifics of this protocol, however, we wish to highlight the benefits of dynamic load balanced I/O scheduling, multicast messaging (in the control plane), and reservation-based payload delivery – for storage clusters.

This paper leaves out of scope the upper layers of the storage stacks that include storage access layers (block, file and object). The mechanisms we describe apply to coarse grained object storage primitives: getting and putting chunks. A chunk is the finest grained unit that the system retrieves from or stores in its distributed persistent storage. A chunk, or a “block”, contains user payload and/or user or system metadata. Replicast distributes chunks optimally and safely, using a combination of consistent hashing and dynamic load balancing.

978-1-5090-3315-7/16/$31.00 ©2016 IEEE

A. Problem Definition

Fully distributed storage clusters have many initiators and many storage targets. There are too many of each for the cluster to be controlled from a single active node.

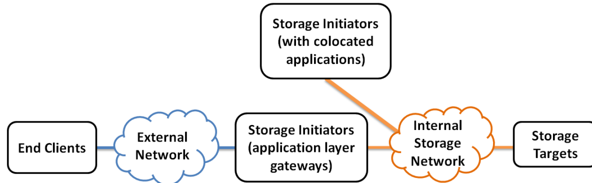

Fig. 1. Distributed Storage Cluster

The initiators may be collocated on the machines that also host the end clients, or an initiator may act as an application layer gateway providing a storage service access point (for block, file or object APIs) and bridging between an external network and the internal storage network. Each storage target also acts as an initiator when replacing lost replicas.

In a large fully distributed storage cluster processing large numbers of storage transactions every second in its Internal Storage Network (Figure 1), the following is generally true:

- **Multiple Replicas:** A put transaction must produce multiple chunk replicas on different storage targets (typically, 3).
- **No Location Tracking Metadata:** Initiators must be able to retrieve a chunk at a later time without requiring location tracking metadata.
- **Chunk Maintenance:** Lost replicas (of any chunk) or whole servers must be replaced by copying surviving replica(s), preferably automatically.
- **Earliest Transaction Completion:** We want the earliest transaction completion (sufficient number of target servers acknowledging the chunk after its payload has been verified and persistently stored).
- **Chunk transfers are relatively short in duration**. Mid-transfer rate adjustments do not speed up the I/O sufficiently to justify added complexity and overhead.
- **Traffic is sparse**. At any given instant only a small fraction of potential node-to-node flows are active.
- **Traffic is effectively random**. The current set of active flows will be uncorrelated with the set of active flows one second later.

It is our belief that for storage clusters where all of the above applies, new Initiator ⇔ Target storage protocols are required and are warranted.

B. Replicast

Replicast [2] is commercially available as part of the NexentaEdge object cluster [3, 31]. The name is derived from “Replication multicast/unicast”, although the term simultaneously happens to be a full anagram of the word “particles.” Replicast’s “particles” are, effectively, chunks of data and metadata.

Replicast is a layer “4.5” protocol: it is above the traditional transport layer 4 (TCP, UDP) but is still functioning below the traditional application layers (5 through 7). Other protocols including iSCSI and iWARP (RDMA over IP) have filled this 4.5 layer that was not anticipated in the original OSI model.

Replicast multicast negotiations allocate network bandwidth, target IOPS and target persistent storage capacity. Replicast does not require any node of the cluster to have complete knowledge of the resource allocations. Initiators only know about the transactions they are initiating. Each storage target only knows about its own resources and commitments. There is no need for all data to be collected at one central planning node: just as with a market system, the interactions of players with limited knowledge can still converge on the correct answer even at vast scale. Research has shown that market-style algorithms can optimally allocate resources even in the absence of an actual marketplace [4].

The rest of this paper is organized as follows. Section II introduces Replicast in the context of scalable distribution of content using load balanced target selection. Section III surveys flow scheduling techniques in a data center. As opposed to conventional TCP or similar connection oriented transports, Replicast uses bandwidth reservation to allocate network resources. Section IV will measure, analyze and evaluate Replicast performance and scalability relative to unicast protocols.

II. Load-Balanced Target Selection

The history of how storage clusters have met increasing demands to scale can be understood in terms of their networking. In the previous decade, pNFS [5, 6] decoupled metadata from the payloads: pNFS metadata servers fully control target selection, but only require metadata bandwidth. Payload transfers are offloaded to other links and servers. Object-based pNFS [6] and the early object storage systems (Google File System [7] and Hadoop Distributed File System [8]) limited the location tracking metadata to the servers, offloading tracking of the exact location of each chunk to each storage server.

Storage clusters with explicit location tracking metadata can make near-perfect I/O load balancing decisions in real time. The issue we address is how to maintain load balancing when the storage cluster scales to such an extent that maintaining and accessing

location-tracking metadata is no longer feasible. We also consider whether it makes sense to sacrifice load balancing in order to support linear scale-out via uniform distribution. This paper presents a certain perspective and definitive answers to these persistent questions.

A. *Consistent Hashing is Too Consistent*

Consistent hashing was invented in the late 1990s and popularized for storage clustering by Amazon's Dynamo [9]. It combines a uniform hash, preferably a cryptographic hash, of the content with hashing storage targets to a "hash ring". Chunks are assigned to the next *n-replicas* targets in different failure domains moving clockwise through the ring. A key benefit of consistent hashing is that an add or drop of a target from an *n* node cluster only reassigns the location of 1/nth of the chunk replicas.

Consistent hashing (CH) algorithms and derivatives are used by the majority of recently produced distributed-storage solutions including Ceph [10] and OpenStack Swift [11]. Initiator driven allocation (whereby a storage initiator (Figure 1) uses CH to select storage targets) eliminates the need to maintain, track and centrally manage chunk/block location metadata. The requirement to generate the same location(s) on a later get (read) means in turn that those selected locations (targets) cannot be influenced by the dynamic factors such as runtime utilizations of storage targets. This is true of even a very sophisticated consistent hash algorithm such as Ceph's CRUSH, which factors in a static cluster's topology (CRUSH map).

Any valid consistent hash algorithm must produce results that are statistically random, because any detectable pattern could be exploited for a denial-of-service attack (that could then deliberately misbalance the load). Random or pseudo-random selection, however, should be distinguished from load-balanced distribution. The single most likely result of flipping a coin 1000 times is indeed 500 heads. However, the probability of getting exactly 500 heads is only 2.522501818% [12]. Replicast also hash-assigns each chunk. However, (and this is its crucial difference from all existing consistent hash implementations) Replicast hashes to a sub-group, the "negotiating group." It subsequently load-balances within each independent sub-group. Replicast does not track which storage targets in the negotiating group have chunk replicas. Therefore, background maintenance does not require metadata updates (a big plus). Chunks are located within the negotiating group using multicast messaging.

Dividing the storage cluster into multiple multicast groups limits the control plane traffic that any one node receives but still provides load-balancing. (By contrast, broadcasting requests to all nodes would produce far too much control plane traffic.) The optimal size of a Replicast negotiating group varies – for deployment and testing we so far favored three times the default replication count (9). Picking the best 3 out of 9 storage targets will produce a better selection than selecting the best 3 of 6. Doubling the size of the negotiating group will also double the number of control plane requests that each node in the group receives, but it will not double the quality of the target selection. Of course, all control plane bandwidth reduces the bandwidth available for data.

B. *Replicast Put Transaction*

Replicast prevents congestion drops by governing the source of every frame feeding the network queues used for its traffic class. Replicast transactions take place on an isolated network that only carries Replicast traffic: the network may be physically separate or a VLAN with its own L2 traffic class.

For a put transaction, an initiator multicasts a put request specifying the cryptographic hash and size of a to-be-put chunk to the negotiating group. In response, each storage target in the group unicasts a "bid" response indicating, among other things, the window of time the chunk can be accepted (note that the description in this section omits details related, for instance, to distributed deduplication, capacity management and error handling).

Each bid is extended by a configurable amount (percentage of the minimal reservation required to process a given chunk size) to allow the initiator to find a common sub-window from multiple bids. Each bid also implies that the corresponding target has (tentatively) committed its resources to receive the chunk within the given time window. The initiator evaluates the collected set of response bids and then makes a selection of the set of targets to receive the chunk. This subgroup of selected targets is termed the "rendezvous group". A pre-selected rendezvous group may be dynamically configured for the selected membership. However, to avoid any need for custom switch firmware it may be easier (or more practical) to select a pre-configured multicast group that has the desired membership. Independently of how the rendezvous group was selected, its address and its membership are included in the put accept message multicast to the negotiating group.

The initiator then executes the rendezvous transfer at the agreed time. In response, each member of the rendezvous group acknowledges the complete chunk transfer. There are corner cases, of course, when this sequence will not result in creating sufficient replicas of the chunk – the Replicast initiator then will simply retry the transaction. Figure 2 shows one specific Replicast interaction with initiators A, B, and C attempting to put a chunk to negotiating group Y. From the perspective of one member of this group denoted as server X, the corresponding put requests arrive in close succession. First, initiator A executes a put-request for the chunk

0x8fe1b and receives a bid indicating that the server X is available right away.

Second, initiator B that has lost this particular "race" to A for its own chunk 0xa428c also transmits a put control message and receives a bid from the same server X. The difference though is that the second bid is shifted 120us into the future – the time that is defined (in this example) by the width of the previous reservation for the chunk 0x8fe1b.

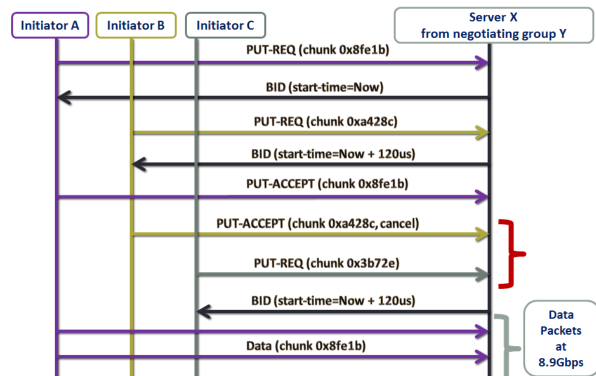

Fig. 2. Replicast Put Transaction (example)

In the center of Figure 2, several things happen. Initiator A accepts server X's bid. Initiator B's accept message (multicast to group Y) does not include server X, which translates as a *cancel* for the corresponding server X's bid. Server X in response will trim its reservation for A to the specified duration, and cancel its reservation for B. Initiator C then sends yet another put-request. At this point, the server X will have released its tentative resources reservation for B and granted a new tentative reservation for C, quite likely for the same time window previously reserved for B. Due to the multicast nature of this control plane, same or similar interactions play out in parallel with each member of the negotiating group Y.

Notice a certain inevitable tradeoff of the reservation based protocol, denoted with the curly bracket on Figure 2. Let's assume a put request from C for its chunk 0x3b72e arrives prior to B's cancelation. In this case the server X's bid that C receives will have to account for both A's and B's reservations. And even though initiator B will (in this example) cancel its bid, initiator C will have to execute based on the information that server X can only offer a time window delayed into the future by, effectively, two put requests. Server X could, of course, eliminate this scenario by waiting for the put-accept message from B prior to responding to C. The model that we describe in this paper, and the NexentaEdge product [3], both favor zero latency as far as control plane responses.

C. *Retrieving Chunks*

Replicast uses multicast messaging to eliminate the need to store location tracking metadata. A chunk is retrieved from a negotiating group as follows:

- The initiator multicasts a get request to the negotiating group for the desired chunk.
- Each storage server in the group that holds a replica of the chunk responds with a bid on when it (the server) could deliver the chunk. Those that do not have the chunk so indicate.
- The initiator then multicasts an accept message to the negotiating group specifying which server should respond. The initiator does not need to wait for all responses when fetching a payload chunk.
- The selected server sends the chunk to the initiator.

This paper leaves out of scope optimization mechanisms that would have storage targets (lazily) sharing their selected server sets to enable more narrow multicasting to the probable locations. For payload chunks false positives could easily be retried (this type of failure should be rare as it implies that all redundant copies have been replaced prior to updating or invalidating the cashes). For operations on metadata, the idea is to narrow down last-version related information. For simplicity, both the model and initial implementation use multicasting to the negotiating group to resolve both metadata and payload chunks.

III. Network Utilization

Most distributed storage clusters deployed today "sub-contract" allocation of one of the most critical resources - network bandwidth - to TCP. TCP is, of course, designed for general purpose traffic. Chunk transfers (chunk-flows) within a large fully distributed storage cluster have a number of characteristics that call for certain specific congestion controls. Flow scheduling must be tailored as well to relatively short flows between pseudo-randomly selected endpoints. The fact that each and every stored or to-be-stored chunk has a known size can be used to deterministically optimize storage driven network communications.

In this section, we first survey works related to flow scheduling and bandwidth reservation. Most propose flow scheduling better suited to storage than generic TCP. We will then discuss our preferred solution, Replicast congestion control and scheduling.

A. *Survey of Flow Scheduling Techniques*

TCP/IP is the most widely deployed method of allocating network bandwidth. Numerous studies have shown the shortcomings of the classic TCP algorithm for dealing with storage traffic. However, TCP congestion control is not a static algorithm; it is constantly improving [13]. This includes using more

sources of information for the network congestions state and improving the rate adjustments made when congestion is detected.

Datacenter TCP (DCTCP) [14] provides an improved algorithm to determine congestion by tracking ECN marking. Glenn Judd's analysis of DCTCP [15] cited issues with DCTCP sharing a network with TCP, but those concerns would not be relevant for storage clusters. It merely requires the use of a distinct storage traffic class.

Timely [16] is a similar congestion control algorithm which measures network congestion by using precise timing to detect changes in packet round trip times, rather than relying on switch ECN marking. Survey of data center flow scheduling schemes [17] discusses several non-TCP solutions. Many of these, such as Rate Control Protocol (RCP) [18], require special support from network routers and/or switches. Such modifications are infeasible for commercial deployment. Other schemes, such as Deadline Aware Queue (DAQ) [19], have merit for general purpose data center traffic that include very long flows. However, they adjust rates during overlapping flows. With storage clusters, each chunk transfer is short lived. A simple scheduling algorithm that schedules the entire chunk transfer is all that is in fact required.

DCQCN [20] is proposed for RoCE [21] networks that incorporates the DCTCP algorithm with the QCN (Quantized Congestion Notification) protocol developed for IEEE 802.1 DCB (Datacenter Bridging) [32]. RoCE is RDMA over Converged Ethernet, effectively a port of InfiniBand over Ethernet. DCQCN adds the benefits of in-path congestion notifications with round trip congestion notification. OpenTCP [22] proposed tracking congestion on a datacenter-wide basis for Software Defined Networks and then using this information to tune TCP on each sender with a custom kernel module. OpenTCP relied upon a central "oracle" to collect/process network status – a potential impediment to scalability. Given relatively uniform distribution of traffic in clusters local measurement of congestion across all flows could provide a reasonably accurate estimate of overall network congestion.

None of these techniques change the fundamental contention-based sequence: pick an initial speed, transmit, measure congestion, adjust speed and repeat. Ongoing developments in the space improve measurement of congestion, adjust the initial windows or improve the additive increase and/or multiplicative decrease to be more intelligent. These are improvements over early implementations but they still must generate congestion to adjust rates. The alternative to "congest-then-adjust" strategy is to reserve network bandwidth before it is used. A survey of flow scheduling schemes in data centers [17] defined several classifications for the scheduling algorithms. According to the classification [17] of flow scheduling schemes, Replicast is a non-TCP based deadline-aware protocol without requirements for special hardware or software on the switch (Figure 3).

Replicast is deadline-aware because every chunk-flow has a known size (size of the chunk) and will finish by a certain negotiated time. Replicast can also be appropriately labeled "edge based" since the clustered storage targets negotiate non-overlapping time slots for each chunk-flow, which also means that the flows do not interleave on the edge links. It is "one-shot": chunk transfers do not require rate adjustments mid-transfer.

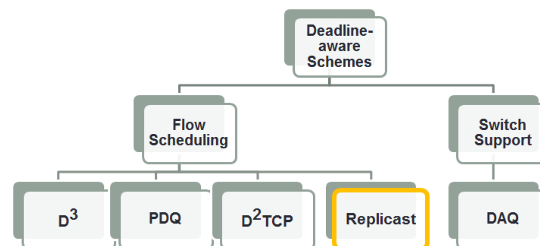

Fig. 3. Part of flow scheduling classification [17]

The only required switch support is limited to no-drop Ethernet [34], which is common for enterprise class switches. The following table compares where and how congestion control is performed:

TABLE 1. WHERE/WHEN CONGESTION CONTROL IS DONE

Protocol	How Often	Where
Rate Control Protocol	Per Frame	Switch
DCTCP	Per Frame	Target OS stack
Replicast	Per Chunk	Replicast userland at storage target

B. *Reservation Based Delivery*

Reservation based protocols execute extremely efficiently during the reserved time windows. There are inevitable idle intervals, though, in-between. The time required to set up a reservation is determined by the round trip time – efficiency of reservation based control depends therefore on having chunks larger (on average) than a certain threshold.

If the gaps between reserved transmissions are small, the efficiency of reservations will be higher than contention algorithms can achieve. Stream oriented reservation protocols configure each switch on the path to support the flow. This does not work for chunk transfers. For example, supporting per chunk reservations for 128 KB chunks would require tracking tens of thousands of reservations per second for each storage target on each switch along the path.

Replicast does not dynamically configure switches for each chunk transfer. The switches are configured to support all possible flows. Senders are only permitted

to transmit when they have a reservation granted by each recipient. This is enforced by shutting off network access for non-compliant nodes. Dynamic switch enforcement is not required. Relying on nodes obeying the reserve-first rule eliminates the cost of dynamically reconfiguring switches but still requires the reservation granter to understand all potentially competing flows. Replicast targets continuously track their own flows and grant subsequent reservations based on this tracking information: no cluster-wide knowledge is ever needed.

C. Captive Congestion Points

Replicast exploits specific network topology including a non-blocking core. Figure 4 illustrates three potential congestion points for a data center storage cluster:

- **Initiator**: The initiator can apply rate shaping to limit transmitted traffic to the desired rates. This would also avoid periodic bursting of packets, providing the same benefit addressed with Paced TCP [23].
- **Non-Blocking Core**: The core network can be provisioned as a non-blocking core.

Which leaves:

- **Storage Target**: In a distributed storage cluster, each target can at any point in time be selected by a random number of other clustered nodes. The egress port to the storage target is, therefore, the sole source of congestion (Figure 4).

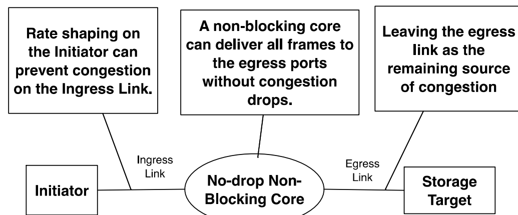

Fig. 4. The three potential congestion points

Storage protocols, therefore, must be explicitly designed to manage (or totally prevent) egress congestion. In this paper we advocate one way: using reservations. Replicast targets issue reservations for their own CPU/disk resources and link bandwidth: one reservation per chunk flow.

D. Replicast Bandwidth Allocation

The Replicast protocol divides link capacity through each captive congestion point between contention allocated and reserved bandwidth, and the contention (aka unsolicited) bandwidth is used to create reservations. The link capacity for the storage traffic class is divided between reserved (data) bandwidth and contention (control) bandwidth. Replicast transfers payload chunks using the reserved bandwidth at the negotiated rate. The sender does not pause waiting for disk I/O to complete, nor does the recipient slow the flow because it is running out of buffer space. Chunks are transmitted from RAM to RAM. The reserved bandwidth is therefore 100% utilized once the transfer is initiated.

The sustained unsolicited rate must be lower than the allocated unsolicited rate to allow for random variation. The goal is to be confident that a surge in unsolicited traffic will not result in a congestion drop. No-drop Ethernet [34] marshals switch resources to manage brief durations exceeding the sustained rate, and eliminates any need to reserve specific time slots – the service will resolve issues as long as the aggregate of reserved and unsolicited frames does not exceed available network buffering.

The (configurable) unsolicited rate must take into account the number of unsolicited packets per each chunk transfer, the size of the negotiating group relative to the rendezvous group, average chunk size(s) and read/write ratios. It is easy to come up with the worst case estimation; questions of safe enough unsolicited minimum as well as its runtime adjustment – those are questions beyond the scope of this paper.

E. Rendezvous Transfers

Replicast bulk payload transfers are called “rendezvous transfers”. They may be performed using multicast messaging to multicast groups which represent a specific subset of a negotiating group or using unicast addressing to a specific target. The variant using multicast payload delivery is referred to as “Replicast-M” (“M” for multicast).

Rendezvous transfers are performed after control plane exchanges have established a reservation granted by each of the receivers. The transfers occur at the provisioned rate for reserved payload. Senders can “floor it” immediately. The feasibility of using unreliable datagrams to provide reliable pre-negotiated transfers was demonstrated for MPI libraries using InfiniBand Unreliable Datagrams [24, 25]. Replicast extends this concept to multiple deliveries.

F. Replicast-Hybrid

The Replicast-H variant of the Replicast protocol (“H” for hybrid) combines multicast control plane with unicast delivery. The hybrid solution is desirable when, for instance, multicast rendezvous groups cannot be dynamically configured or preprevisioned in sufficient numbers. With this variation, a put transaction performs the following steps:

- Similar to Replicast-M (Figure 2 above), Replicast-H initiator multicasts a put-request to the group, identifying the chunk, its compressed size and the earliest time this initiator can start transmitting the chunk’s payload.

- Each storage target in the group not already holding the chunk responds with a bid committing its resources and its bandwidth for the specific window of time in the future.
- The initiator collects the bids and tries to select multiple consecutive time windows for the subsequent unicast transmission(s).
- The initiator then unicasts the rendezvous transfers at the specified time(s).
- The selected targets each reply with a Chunk ACK after it successfully saves the chunk to persistent memory, or earlier in the event of any error.
- The Initiator keeps repeating until the total number of required replicas have been put.

Both Replicast-M and Replicast-H strategies rely on reserved bandwidth and use a single acknowledgement for an entire chunk transmission. A cryptographic hash on every chunk verifies error free reception (and later – retrieval) and also detects lost packets, however rare. While the cost of full chunk retransmission is high, the judgment call that we had made previously (based on a realization that, for instance, traffic shaping in modern data centers effectively prevents network congestion drops) proved to be feasible and working.

G. *Reliable Multicasting*

Replicast uses unreliable datagrams. The general problem of reliable multicasting has been well studied [26]. However, reliable multicasting in a large distributed storage cluster (the use case that we describe here) – is a far more specific problem:

- Multicast delivery is to a handful of targets. There is no “ACK implosion” problem from too many targets acknowledging the transaction.
- The solution is constrained to networks with extremely low physical error rates with reservations to prevent congestion drops.
- Each chunk transfer is short, with a length known before the transfer begins. No targets attempt to join an in-progress delivery.
- Each chunk transfer can be verified by a cryptographic hash of the chunk content.

Therefore, a simple acknowledgement of each chunk transfer, with upper layer driven retries, is all that is required to achieve reliable chunk transfers.

H. *Multicast Delivery is Efficient*

Multicast delivery is efficient: the initiator only sends once to reach multiple targets. The authors’ initial object storage design [27] sought to reduce the network overhead of replication from three copies to two by hybridizing network replication with local replication. The prospect of zero extra transmissions motivated our initial exploration of multicasting.

There's always a cost and associated tradeoffs in almost any new design. Multicast delivery, for instance, depends on the capability to find those 3 servers that provided bids, intersection of which yields the ultimate window of time sufficiently wide to execute the chunk. This in turn motivates provisioning of extra-width into each and every reservation, to increase the corresponding chances. But each additional microsecond in the reserved windows increases the chances to delay subsequent reservations. As we progressed with the design we found that, even excluding multicast data transfers, using multicast load-balancing and reservations still produces great benefit.

IV. DISCRETE EVENT SIMULATION

Our lab storage clusters have been running for some time, but they unfortunately do not scale anywhere near as large as Replicast is designed for.

To address this limitation, we developed SURGE – a discrete event simulation framework written in Go [29]. Complete and documented source code that includes the framework and concrete models (including basic Replicast) can be found at https://github.com/hqr/surge. The results cited here were generated with the build *d92018f* using *bench.sh* script, also checked in.

Behind the gateways/initiators, there are storage servers. Together, gateways and servers are referred to as nodes and form a distributed cluster. Each simulated node fully owns its configured resources: disk and network interface, and runs in a separate “goroutine”. The latter is a first-order primitive of the Go language that multiplexes potentially hundreds of thousands of goroutines onto OS threads. SURGE therefore is a fully preemptive modeling framework that employs all processor cores of the host to drive its models forward. Each modeled node connects with all other nodes via a pair of Go channels. SURGE’s event types include data events that carry chunk payload and unsolicited control messages.

Each simulated control and data event is precisely timed, with the system time “ticking” in nanoseconds. At each next tick the framework itself makes sure that the modeled NOW does not advance until all the events scheduled at NOW do in fact execute. The simulated datapath uses 9000 byte jumbo frames. The configured bandwidth is reduced by 1% for overhead. Each data frame is a timed SURGE event. Initiators are most often Application Layer Gateways (ALG) accessed over a client network by the end user using standard protocols. The impact of the client network and its protocol selection is not modeled.

We only model put performance. The mix of get versus put transactions, and their subtle performance interactions (particularly related to caching) would require more complex modeling and would force the

simulation to make many assumptions that users might feel are unrepresentative.

A. The Common Model

The following model is henceforth assumed:

- All objects are decomposed into metadata and payload chunks.
- Metadata chunks (typically one per object version) are indexed by the object name and version.
- Payload chunks, which are ultimately indexed by a cryptographic hash of their content.
- There is a non-blocking no-drop network core that connect a large number of edge ports. This simulation models those ports as 10 GbE ports.
- All storage payload transfers occur on a protected no-drop VLAN.
- Transmission errors are sufficiently rare that they do not need to be modeled.

The simulation (with results presented in Section "Results" below) only models put transactions. In actual deployments, concurrent read and writes impact each other. We did not simulate the reads (yet) at least in part because (at the time of this writing) there is no reason to believe that there would be an appreciable benefit of Replicast reads over conventional reads.

As of the time of this writing, SURGE simulations model each initiator working on a single chunk at a time while NexentaEdge implementation begins negotiations for the next chunk while finishing the current chunk. The simulation can of course achieve an extremely high level of parallelism by simply adding more initiators. This would also be effective for a real world storage cluster as well but would not be cost free.

B. UCH-CCP

A fair comparison requires simulating unicast congestion control that was equal to the best TCP congestion control algorithms. Rather than investing in simulating a moving target, we modeled an idealized unicast congestion algorithm that is better than any actually deployed. Specifically, this hypothetical algorithm dubbed "UCH-CCP" (Unicast Consistent Hash using Captive Congestion Point) features:

- Consistent hashing for target selection.
- Unicast UDP that is used for both "connection" setup and for chunk transfers.
- Dynamic reservations for payload transfers, granted by the targets at connection setup times.

UCH-CCP relies on the reserved traffic class to isolate it from non-storage traffic. This is a unicast dynamic bandwidth reservation protocol. It allows the sender to "floor it" after a single round trip. To avoid any biasing, we have even compared against UDP, with its lower overhead per packet, rather than TCP. Figure 5 illustrates the UCH-CCP pipeline stages:

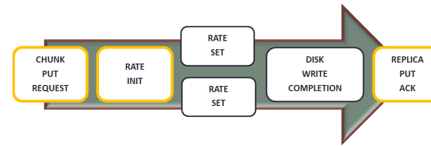

Fig. 5. UCH-CCP Pipeline

"Chunk Put Request" initiates the flow. "Rate Init" response form the target carries the initial rate computed as proportion of the target's bandwidth given the current number of flows (to this target). "Rate Set" represents additional optional rate adjustments (by the target) during the flow. Once the entire chunk is transferred, it is written to persistent storage (signaled by "Disk Write Completion") and then acknowledged ("Replica Put ACK"). UCH-CCP (serial) variant could assign all egress link bandwidth to one chunk at a time. The UCH-CPP model [30] and simulated benchmarks that we show in this paper use an interleaving strategy to more closely resemble TCP/IP (the transport used today by great majority of storage clusters).

Figure 6 illustrates a UCH-CCP interaction with two initiators attempting overlapping chunk puts to the same server. Initiator A receives the full rate (9.9 Gbps) when it starts.

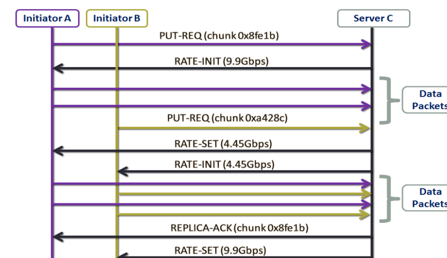

Fig. 6. UCH-CCP protocol (example)

When initiator B requests to put a chunk it receives an initial rate of 4.45 Gbps while initiator A's rate is cut to the same. Once initiator A's chunk transfer completes, it receives its Replica ACK and initiator B's rate is increased to 9.9 Gbps.

C. Replicast

Figure 7 illustrates the put-chunk I/O pipeline of the Replicast-H and Replicast-M protocols. These control messages exchanged as part of each and every put transaction were previously discussed in Section "Replicast Put Transaction".

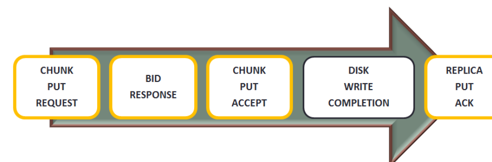

Fig. 7. Replicast Stages

The Replicast simulations model UDP over IPv6. The switch is assumed to have multicast forwarding tables large enough to allow all required rendezvous groups to be pre-configured.

D. *Replicast-H(ybrid)*

Replicast-H combines multicast negotiations with unicast delivery. This provides the benefits of multicast negotiations and use of reserved bandwidth for chunk transfers, but sacrifices the benefits of multicast delivery. As with UCH-CCP the link is still fully utilized during a chunk transfer.

However, unicast delivery may be the preferred solution when the network infrastructure cannot support the required number of rendezvous groups.

E. *Replicast-M(ulticast)*

Replicast-M is modeled with static rate control for unsolicited traffic. Each Initiator limits generation of unsolicited requests to a configured rate. This rate is set so that there is high confidence that the actual aggregate unsolicited bandwidth will be below the bandwidth set aside for unsolicited traffic. Rendezvous groups may be dynamically configured. One method is to dynamically specify the rendezvous group as the subset of the negotiating group. The BIER (Bit Indexed Explicit Replication) [28] protocol under development within the IETF allows multicasting to a subset of a "BIER Domain" to be identified by an explicit bitset. Replicast can map each negotiating group to a BIER domain and specify the targets of each rendezvous transfer with a bitmask.

Alternately, rendezvous groups may be pre-configured and dynamically selected. IPV6 MLD is used to pre-configure every possible 2 or 3-member subset of each negotiating group. The initiator can select the pre-configured group because it has an exclusive reservation for each of its members. Pre-configuring rendezvous groups does limit the scale of one cluster. If a negotiating group has 9 members, then 120 rendezvous groups can enumerate every 2 or 3-member subset (9*8*7/3*2 + 9*8/2 = 120). Supporting 900 storage targets in a cluster and 9 targets per group would then require 120 * (900/9) multicast forwarding rules in each switch. Growing beyond 12,000 targets would quickly exceed the table capacity for virtually all existing switches and thus require either special switches or partitioning of the storage cluster into multiple federated clusters.

F. *Results*

Typical simulated benchmark took us anywhere between 90 minutes and 6 hours on a 24-core system, with every transaction timed and logged for later analysis. These results correspond to a non-blocking 10GE core connecting 30 or 90 initiators putting to 90 targets. Simulated CPU is effectively unlimited. In the benchmarks, initiators generate chunks (as fast as they can) and store 3 replicas of each chunk on the targets.

Table 2 summarizes the results for 90 targets and 90 initiators for 16K, 128K and 1MB size chunks at disk throughputs 400MB/s and 1000MB/s, respectively. Chunk sizes are shown in the left column. Each cell of the table contains 3 types of averages (means): performance (chunks/sec), utilization of target disks (%), and chunk put latency (microseconds).

TABLE 2. 90 INITIATORS, 90 TARGETS, 400MB/S DRIVES

Chunk	uch-ccp	replicast-m	replicast-h
16K	536,950 c/s 70.6% 164.7us	635,750 c/s 83.6% 139.0us	525,100 c/s 69.0% 168.4us
128K	58,400 c/s 66.4% 1401.7us	80,700 c/s 90.6% 1039.8us	73,400 c/s 83.6% 1132.6us
1M	6,733 c/s 69.6% 10260.2us	8,067 c/s 85.6% 9210.8us	8,000 c/s 84.9% 8945.9us

The corresponding aggregate performance and utilization numbers for each modeled protocol are listed as well. Even for smaller size chunks, Replicast-M achieves better latency and throughput than the other two protocols. Figure 8 can help visualize the put-chunk latencies for both Replicast-M and UCH-CCP. Neither protocol creates any notable steady-state patterns (such as the TCP sawtooth).

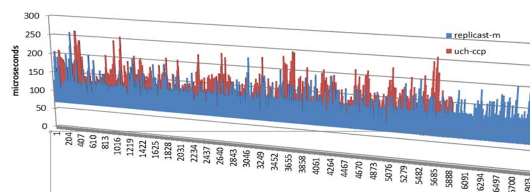

Fig. 8. Put latency – 30 initiators, 90 targets, 16 KB Chunks, 400 MB/s drives

The X axis on Figure 8 represents successive chunk numbers from 1 to the maximum (number) performed during this fixed-time benchmark. Figure 9 summarizes the throughput for Replicast-M compared to UCH-CCP and Replicast-H, for both 400 MB/s (left) and 1000 MB/s (right) drives.

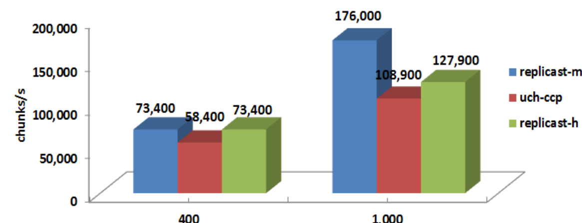

Fig. 9. Put throughput comparison for 128K chunks

Replicast-H and Replicast-M show similar performance with the slower drives but only Replicast-M scales almost linearly with the faster drives: 2.18 increase in the throughout (relative to the 2.5x increase in disk speed). Replicast-M scales even better for 1MB chunks: 2.4 increase. The unicast protocols, meanwhile, show less improvement, indicative of network bottlenecks. On the other hand, the larger control plane overhead is in part responsible for the Replicast-M performance with 16 KB chunks.

Table 3 compares the impact of faster disk drives on transactional latency. Similar to Table 2 above, the results include average performance (chunks/sec), disk utilization and put latency (microseconds). If the network caused zero delays, the expected latency with 1000 MB/s disks would be 0.4 of the latency with 400 MB/s disks. Replicast-M scales extremely well for medium and large chunks, while Replicast-H outperforms its unicast counterpart for medium and large chunks. Both Replicast "flavors" are clearly impacted by the control plane overhead for 16 KB chunks.

TABLE 3. 90 INITIATORS, 90 TARGETS, 1000MB/S DRIVES

Chunk	uch-ccp	replicast-m	replicast-h
16K	918,500 c/s 48.1% 96.0us	907,400 c/s 47.4% 97.0us	632,200 c/s 32.9% 139.3us
128K	108,900 c/s 47.0% 784.7us	176,000 c/s 74.7% 489.5us	127,900 c/s 55.0% 673.0us
1M	13,475 c/s 48.4% 6042.9us	19,850 c/s 68.6% 4242.0us	16,075 c/s 57.1% 5163.8us

G. Lab Testing

NexentaEdge object cluster [3, 31] has been available for beta users since late 2014. This solution implements the Replicast protocol in user mode Linux. Table 4 below shows NexentaEdge block throughput performance for 16K chunks:

TABLE 4. NEXENTAEDGE PERFORMANCE – 16K CHUNKS

Chunk/s	23351.91
MB/s	396.12
Write Latency (us)	2.4505

- 2 application layer gateways (aka initiators)
- 10 storage targets, each with 128 GB RAM, 10x4 TB HDDs, 2x800 GB SSDs, 10 GbE Ethernet
- Mellanox SX1024 switch

Results that we get in our lab are consistently better, often 2 to 7 times better, than the performance of Ceph and OpenStack Swift storage clusters on the same (apples-to-apples) hardware configurations.

H. Replicast Strengths and Weaknesses

The simulation results confirm one of the primary design goals of the Replicast protocol: better load-balancing for both network bandwidth and target IOPS results in higher storage backend utilization. As stated in the Section "Problem Definition", Replicast is designed for a very specific set of workloads and was never intended to perform as a general purpose transport. From the general purpose transport perspective, Replicast does have weak points:

- Transfers must be for known-size non-gigantic chunks.
- The storage cluster membership must be enumerated and have short cluster trip times.

Other shortcomings are evident in the simulation results. As per discussion in Section "Load-Balanced Target Selection", tentative reservations which are not ultimately accepted can delay bids for later put requests. This reduces storage backend utilization and increases transactional latency. For instance, given 176,000 chunk/sec average throughput for the 90 targets assembled in 10 groups (see Table 4, Figure 9), the cluster-wide chunk arrival time would range between 25us and 75us. Assuming medium 50 microseconds, the Poisson lambda for a storage server would equal about 0.02. Thus, for any storage server processing 128K put-chunk workload in this (90 initiators, 90 targets, 1000MB/s drives) configuration, the probability of receiving a new put request during the next 7 microseconds (the time required to send the bid and receive a response) computes as:

$$\boldsymbol{f(t) = 1 - \lambda e^{-\lambda t}} = 13\%$$

Table 5 summarizes the probabilities for any given target to receive a put while the prior reservation is still tentative (that is, not yet accepted or canceled). Delayed reservation from a single server does not necessarily mean that the corresponding chunk will have to be sub-optimally delayed: for this to happen and given the group size 9, more than 6 servers of this group must be delaying this particular reservation as well. Clearly though, higher ratios of back-to-back reservations will adversely affect performance:

TABLE 5. PROBABILITY OF (PUT REQUEST ARRIVING WHILE THE PRIOR RESERVATION IS STILL TENTATIVE)

Chunk	Put interarrival time	λ	Poisson probability
16K	11us	0.09	46.7%
128K	50us	0.02	13%
1MB	500us	0.002	1.39%

As for smaller chunks, Figure 10 illustrates the comparative performance of the protocols for 16KB

chunks, for 400MB/s drives (left) and 1000MB/s drives (right), where the benefit of Replicast is far lower than for 128KB and 1MB chunks. Longer intra-cluster round trips result in higher probabilities of reservation conflicts. Another potential weakness is that load balancing is limited to the scope of the negotiating group. Although much less likely, negotiating groups can become unbalanced.

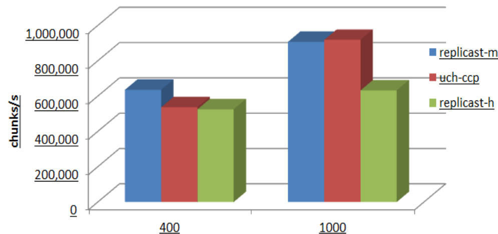

Fig. 10. Throughput: 90 initiators, 16KB chunks

Replicast packets used to set up a flow are larger than those for a TCP connection. Replicast control packets take slightly longer to transmit. All benchmarks in this paper were conducted with 100 byte UCH-CCP control messages and 300 byte Replicast control messages. The overhead (that includes, for instance, cryptographic hash of the chunk) hurts performance when the chunk size is smaller. The results suggest that shrinking the Replicast control packets and/or optimizing the time required to make reservations could lead to further optimizations. This is one area where the size of the negotiating group chosen can have an impact. Larger negotiating groups load-balance better but require each storage server to process more control packets for the same amount of actual work.

I. Next Steps

When covering the topics of I/O performance, it is often difficult to address all pertinent questions at once, simply due to the combinatorial nature of possible configuration/workload parameters. The questions include impact of a) concurrent gets and puts, and their ratios, b) ratios of initiators to targets, c) size(s) of the negotiating group, and more. For the next possible steps or to find answers to at least some of those questions, please refer to [33] – in particular the blogs titled "Choosy Initiator" and "The Better Protocol".

V. CONCLUSIONS

Both conventional consistent hashing and conventional unicast congestion control lead to poor allocation of storage cluster resources. Conventional consistent hashing determines the location of replicas (or stripes) of content without regard to current processing loads. Its pseudo-random selections are not as efficient as dynamically load balanced selection. By contrast, Replicast dynamically selects storage targets and achieves the same lack of central metadata processing as well as balanced resource utilization across targets. Conventional unicast congestion control can never achieve high link utilization because it depends on creating congestion in order to trigger congestion avoidance. Dynamic negotiation of bandwidth reservations can achieve more efficient link utilization.

Results in this paper are supported by NexentaEdge tests in our lab that we run on a variety of hardware for 6 to 10 storage targets, and the SURGE simulations – for orders of magnitude larger configurations. These performance results are consistent with the expectation that known-size, relatively short flows with pseudo-random target selection will benefit from a mechanism and design such as Replicast. Beyond the topic of storage clusters, we advocate the potential benefits of tailoring strategies to specific application requirements.

REFERENCES

[1] Amotz Bar-Noy, Reuven Bar-Yehuda, Ari Freund, Joseph (Seffi) Naor, Baruch Schieber, "A Unified Approach to Approximating Resource Allocation and Scheduling," http://www.cs.technion.ac.il/~reuven/PDF/BarBar.pdf

[2] Alex Aizman, Caitlin Bestler, "Beyond Consistent Hashing and TCP: Vastly Scalable Load Balanced Storage Clustering," http://www.snia.org/sites/default/files/SDC15_presentations/dist_sys/AlexAizman_Beyond_Consistent_Hashing_and_TCP.pdf

[3] "NexentaEdge," https://nexenta.com/products/nexentaedge

[4] M.A. Gibney, N.R. Jennings, "Dynamic Resource Allocation by Market-Based Routing in Telecommunications Networks," http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.34.1485&rep=rep1&type=pdf

[5] RFC 5661, "Network File System (NFS) Version 4 Minor Version 1 Protocol," https://tools.ietf.org/html/rfc5661

[6] RFC 5664, "Object-Based Parallel NFS (pNFS) Operations," https://tools.ietf.org/html/rfc5664

[7] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, "The Google File System," http://static.googleusercontent.com/media/research.google.com/en//archive/gfs-sosp2003.pdf

[8] "HDFS Architecture Guide," https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html

[9] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels, "Dynamo: Amazons Highly Available Key-value Store," http://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf

[10] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, Carlos Maltzahn, "Ceph: A Scalable, High-Performance Distributed File System," http://www.ssrc.ucsc.edu/Papers/weil-osdi06.pdf

[11] OpenStack, "Swift Documentation," http://docs.openstack.org/developer/swift/

[12] http://stattrek.com/online-calculator/binomial.aspx

[13] IETF, Internet Congestion Control research group, https://datatracker.ietf.org/rg/iccrg/charter/

[14] draft-ietf-tcpm-dctcp-01, “Datacenter TCP (DCTCP): TCP Congestion Control for Datacenters," https://tools.ietf.org/html/draft-ietf-tcpm-dctcp-01

[15] Glenn Judd, “Attaining the Promise and Avoiding the Pitfalls of TCP in the Datacenter," https://www.usenix.org/system/files/conference/nsdi15/nsdi15-paper-judd.pdf

[16] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, David Zats, “TIMELY: RTT-based Congestion Control for the Datacenter," http://conferences.sigcomm.org/sigcomm/2015/pdf/papers/p537.pdf

[17] Roberto Rojas-Cessa, Yagiz Kaymak, Ziqian Dong, “Schemes for Fast Transmission of Flows in Data Center Networks," https://web.njit.edu/~rojasces/publications/royazisurvtu15.pdf

[18] N. Dukkipati and N. McKeown, “Why flow-completion time is the right metric for congestion control,” https://www.eecs.berkeley.edu/~sylvia/cs268-2014/papers/rcp-ccr.pdf

[19] C. Ding and R. Rojas-Cessa, “DAQ: Deadline-Aware Queue scheme for scheduling service flows in data centers,” https://web.njit.edu/~rojasces/publications/coroicc14.pdf

[20] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, Ming Zhang, “Congestion Control for Large-Scale RDMA Deployments," http://conferences.sigcomm.org/sigcomm/2015/pdf/papers/p523.pdf

[21] RoCE, http://www.roceinitiative.org/

[22] Monia Ghobadi, Soheil Hassas Yeganeh, Yashar Ganjali, “Rethinking End-to-End Congestion Control in Software-Defined Networks," http://conferences.sigcomm.org/hotnets/2012/papers/hotnets12-final85.pdf

[23] David X. Wei, Pei Cao, Steven H. Low, “TCP Pacing Revisited," http://people.cs.pitt.edu/~ihsan/pacing_cal.pdf

[24] Lloyd Dickman, “Pathscale InfiniPath: a first look," https://www.researchgate.net/publication/4195836_Pathscale_InfiniPath_a_first_look

[25] Matthew J. Koop, Sayantan Sur, Dhabaleswar K. Panda, “Zero-Copy Protocol for MPI using InfiniBand Unreliable Datagram," http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.110.483&rep=rep1&type=pdf

[26] Sally Floyd, Van Jacobson, Ching-Gung Liu, Steven McCanne, and Lixia Zhang, “A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing," http://www.icir.org/floyd/papers/srm_ton.pdf

[27] Caitlin Bestler, “Versioned De-duplicated Object Storage," http://www.advancedhpc.com/data_storage/unified_storage/zetastor/datasheets/OpenstackSwiftCCOW.pdf

[28] IETF BIER, https://datatracker.ietf.org/wg/bier/charter/

[29] Go Programming Language, https://golang.org/doc/

[30] SURGE – a discrete event simulation framework, https://github.com/hqr/surge

[31] Micron Accelerated NexentaEdge Solution, https://www.micron.com/resource-details/322b8f66-83fe-4aee-9bab-93d3ba267e9a

[32] Data Center Bridging Task Group, http://www.ieee802.org/1/pages/dcbridges.html

[33] Blog storagetarget.com, http://storagetarget.com

[34] DCB Whitepaper, http://www.ethernetalliance.org/wp-content/uploads/2011/10/document_files_DCB_Whitepaper_v2.pdf

VI. Appendix A. Results for 30 Initiators

Figure 11 below shows the sorted chunk put latencies for the smaller (16KB) chunks in the 90x90 configuration (90 initiators, 90 targets). Overall, even for smaller chunks, we see that Replicast-M has consistently lower latency than UCH-CCP.

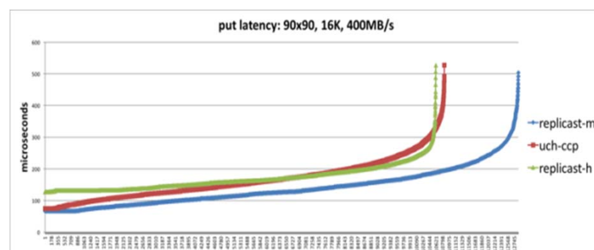

Fig. 11. Sorted Put Latency – 16 KB Chunks

One of the numerous questions we were “asking” the SURGE models was: how will Replicast perform for different ratios of initiators/targets. This section briefly lists two benchmarks for the 1/3 ratio: 30 initiators/gateways generating traffic to 90 storage targets. Figure 12 and Figure 13 illustrate comparative throughput for 128KB and 1MB chunks, respectively, for both 400 MB/s (left) and 1000 MB/s (right) drives:

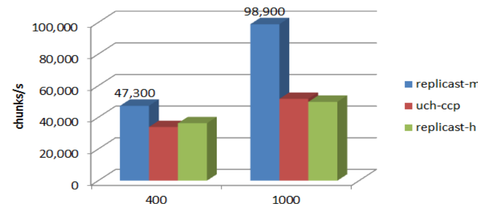

Fig. 12. Throughput: 128KB Chunks, 30 initiators

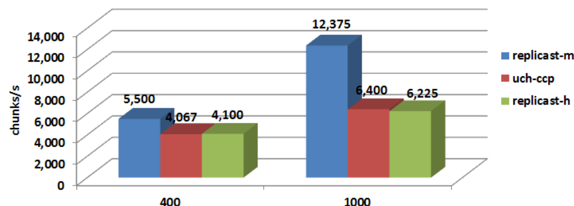

Fig. 13. Throughput: 1MB Chunks, 30 Initiators

Replicast-M scales almost linearly with drive throughput – for medium and large chunk sizes. In this case, 30 initiators are able to keep 90 targets busy enough, achieving higher utilization and hence better throughput. This capability to scale is supported by all our benchmarks, both simulated using SURGE framework and conducted on hardware.