Proactive replication for data durability

Emil Sit, Andreas Haeberlen, Frank Dabek, Byung-Gon Chun, Hakim Weatherspoon
Robert Morris, M. Frans Kaashoek and John Kubiatowicz

ABSTRACT

Many wide-area storage systems replicate data for durability. A common way of maintaining the replicas is to detect node failures and respond by creating additional copies of objects that were stored on failed nodes and hence suffered a loss of redundancy. Reactive techniques can minimize total bytes sent since they only create replicas as needed; however, they can create spikes in network use after a failure. These spikes may overwhelm application traffic and can make it difficult to provision bandwidth.

This paper explores a proactive approach that creates additional copies not in response to failures, but periodically at a fixed low rate. We introduce Tempo, a distributed hash table that allows each user to specify a maximum maintenance bandwidth and uses it to perform proactive replication. Results from a simulation study suggest that Tempo can deliver high durability despite only using several kilobytes per second of bandwidth, comparable to state-of-the-art reactive systems.

1. INTRODUCTION

Most existing distributed hash tables monitor the availability of data and replace lost redundancy on other nodes in reaction to failures [1,4,6,8,18]. The bandwidth needed to support this reactive approach can be high and bursty: each time a node fails permanently, the system must quickly produce a new copy of all the objects that the node had stored [2]. Quick replication is especially important if a DHT is being used in storage intensive applications like OceanStore/Pond [16], OverCite [21], or ePOST [11] where data loss must be minimized. While reactive systems can be tuned to provide durability at low total cost [4], the need to repair quickly can cause dramatic spikes in bandwidth use when responding to failures. In many settings, provisioning for high peak usage can be expensive.

This paper examines the alternative of maintaining availability by proactively replicating objects before failures occur. In particular, we consider constantly creating additional redundancy at low rate. This technique evens out burstiness in maintenance traffic by shifting the time at which bandwidth is used. Instead of responding to failures, a proactive maintenance system operates constantly in the background, increasing replication levels during idle periods. Operating proactively in this manner results in a predictable bandwidth load: node operators and network administrators need not worry that a sudden burst of failures will lead to a corresponding burst in bandwidth usage that might overwhelm the network. Instead, any burstiness in network usage will be driven by the application's actual workload. The question we seek to answer is whether this method can still prevent data from being lost.

This research was supported by the National Science Foundation under Cooperative Agreement No. ANI-0225660, `http://project-iris.net/`. Emil Sit was supported in part by the Cambridge-MIT Institute. Andreas Haeberlen was supported in part by the Max Planck Society.

This paper proposes Tempo, a distributed hash table that uses proactive maintenance. In contrast to systems that use as much bandwidth as necessary to meet an availability specification (given explicitly [1] or in the form of a minimum replication level [6]), each node in Tempo operates under a bandwidth budget specified by the node operator. A budget is attractive because easy for the user to configure: bandwidth is a known, measurable and easily understood quantity. The nodes cooperate and attempt to maximize durability and availability within their individual budgets by constantly creating new replicas, whether or not they are needed at the moment. While systems that specify a number of replicas respond to failures by varying the bandwidth usage in an attempt to maintain that replication level, Tempo instead effectively adjusts the available replication level subject to its bandwidth budget constraints. We show that in simulation based on PlanetLab measurements over a 40 week period, Tempo can maintain more than 99.8% of a 1TB workload durably using as little as 512 bytes per second of bandwidth on each node. With 2K per second per node, no objects were lost: this amount of bandwidth is comparable to that used by reactive systems but Tempo uses this much more evenly.

The rest of this paper is structured as follows: Section 2 provides an overview of the design considerations for a proactive maintenance system. We discuss some aspects of how Tempo would be implemented in Section 3 and evaluate it via simulation in Section 4. Section 5 discusses related work and we conclude in Section 6.

2. DESIGNING PROACTIVE REPLICATION

In such a distributed hash table, nodes cooperate over the wide-area to store a single set of objects. The difficulty in maintaining data is dealing with node failures: the durability and availability of objects are determined by how well the maintenance algorithm can keep up with failures that permanently render data redundancy unavailable.

Certain aspects of the design of a replicated storage system are common to both proactive and reactive maintenance systems. For example, the method for assigning objects to nodes affects the number of nodes that have a copy of each object and thus can potentially participate in creating new redundancy. The format of redundancy (e.g. erasure codes) can also affect resiliency. These questions are discussed in more detail in [4,19,25] among many others.

This section looks at the key question facing proactive maintenance systems: when and how quickly should redundancy be created? We argue that creating redundancy constantly at a limited rate is a simple, flexible, and effective approach to maintain data durably. Our goal in designing

Tempo is to maximize data availability and durability while staying within a bandwidth budget. Given this approach, we consider how to efficiently utilize this bandwidth and what might happen as available storage capacity is consumed.

2.1 Using a bandwidth budget

The question faced by a proactive maintenance system is which repair actions to take and when to initiate them. At any given point in time, the system can take any given object and make a new replica of it. Reactive systems create new replicas after a failure: when the number of available copies of an object drops below some threshold r_L, repair is initiated. When should proactive systems start and stop repair?

One approach for scheduling repairs would be to attempt to predict node failures and increase the replication level of objects that will have low redundancy following a predicted failure. The effectiveness of this scheme would be directly related to the accuracy of the predictor: a string of incorrect predictions can lead to data loss.

A simpler option is to create redundancy as fast as possible until either all disk capacity is exhausted or objects are replicated on all nodes. This corresponds practically to setting $r_L = \infty$ in a reactive system, because such a system will then act as if it was constantly lacking redundancy.

Network bandwidth, however, is not unlimited: while it is typically priced for fixed capacity (e.g., a T1 offers up to 1.544Mbps), pricing agreements may place an upper limit on the total volume transferred in a given period. For example, experiments running on PlanetLab can use up to 10Mbps but if more than 16GB is sent from a single node in a day, its link is restricted to 1.5Mbps and the experimenters are notified. In non-research environments, exceeding the cap may result in additional charges.

In order to use bandwidth when available, but prevent excess, each node n_i in a proactive system should set an outgoing bandwidth cap b_i that appropriately limits the total number of bytes sent per unit time. This parameter represents the amount of bandwidth the node operator is willing to dedicate to data maintenance.

2.2 Impact of budget size on durability

The durability of data in a replicated system can be thought of as the percentage of objects that are not lost after the system has been operating for some time. Constraining the bandwidth available for maintenance will affect the durability that is measured. If the budget is consistently used to create new replicas, one might naturally expect that over time, the number of replicas of all objects would grow without bounds until all available disk capacity was consumed. However, as the number of replicas grows, the number of replicas lost in each node failure grows as well. Thus, if the system runs long enough, it eventually reaches an equilibrium in which the replica creation rate, i.e. the rate at which new copies of objects are being created by the replication algorithm, balances the replica loss rate, i.e. the rate at which copies of objects are lost when nodes fail permanently.

To estimate the position of this equilibrium, we can model the number of replicas of each object using a $M/M/\infty$ birth-death Markov chain. Assume that the system constantly uses its entire budget to repair at rate μ and that it experiences an average per-node failure rate of λ_f. The object is in state r_i when i disks hold a replica of the object (regardless of whether it is online). From a given state r_i, there is a transition to state r_{i+1} with probability μ when $i > 0$ corresponding to repair. There is a transition to the next lower state r_{i-1} with rate $i\lambda_f$ because each of the nodes holding an existing replica might fail. An analysis of this birth-death process shows that the expected state is $\theta = \mu/\lambda_f$. In systems with high available capacity, this value is likely to be larger than the minimum number of replicas required to maintain availability.

We can estimate θ under different scenarios by examining values of μ and λ_f. For example, based on historical measurements of PlanetLab (as shown in Table 1), the average failure inter-arrival time for the entire test bed is 12.53 hours. Since we are interested in the per-disk failure interval we multiply by the average number of nodes in the system (408). Inverting gives an average per-disk failure rate of $\lambda_f = 1/5112 \approx 0.0002$, failures per hour. The repair rate depends on the node storage capacity and the network throughput; the current PlanetLab network capacity is rate-limited to 1.5 megabits per second (Mbps) and each experiment is limited to 5 gigabytes per node. Since it takes approximately 10 hours to transfer 5GB (accounting for some protocol overhead), the creation rate is approximately $1/10$ disks per hour. This results in an equilibrium point of $\theta \approx 511$, which is greater than the average number of nodes present during the measurement period. This means that if the total amount of unique data in the system is less than 5GB, it is possible to replicate each file to all nodes in PlanetLab at 1.5Mbps.

The selection of a correct bandwidth (and hence μ) is important to the ability of the system to prevent data loss. It must be set high enough to keep pace with the average failure rate, but it need not be significantly higher. We show in Section 4 that a maintenance budget of approximately than 2 kilobytes per second is sufficient on PlanetLab, corresponding roughly to $\theta = 7$.

This model is a somewhat unrealistic view of the maintenance process: there can be an infinite number of replicas and there is a transition from r_0 (in which the object is lost) to r_1. However, it conveys the intuition of a how replicas are made and suffices for the purposes of these estimates. A detailed discussion of this and more complete models appears in [5] and [14].

2.3 Budget utilization

Ideally, a node in a proactive maintenance system with a bandwidth limit of b bps would send and receive exactly b bytes every second. This would not only preclude any bandwidth spikes, but also result in the highest number of replicas per object, and thus potentially in the highest durability. However, engineering such a system is not trivial.

Whenever a node finishes downloading a new copy of some object, it must decide which object to download next. Intuitively, it makes sense to give preference to objects with few existing copies, since these are the most likely to be destroyed by future failures. However, the distribution of these objects is not necessarily uniform. As a result, some nodes may end up with significantly more concurrent uploads than others. This has two undesirable effects: first, the upload bandwidth of the other nodes is underutilized, so bandwidth is wasted. Second, the uploads at each node have to share the available bandwidth, so each individual upload takes more time, which in turn increases the chance that more upload requests will arrive before the original ones complete. The result is a 'slow' node with a high number of very slow uploads.

There are several ways to approach this problem. One is to use a set of heuristics (e.g., a limit on the number of concurrent uploads). This approach is conceptually simple, but cannot guarantee perfect utilization. Another way is to associate each node with a fixed set of other nodes from which it can download objects, and to make the association pattern symmetric. This scheme can utilize the entire bandwidth limit, but restricts the set of objects each node can repair at a given time. This approach is the one adopted by Tempo.

2.4 Managing disk capacity

One important concern of a proactive replication system is running out of disk space. To avoid overflowing disks, Tempo allows each node operator to locally bound the total amount of local disk space it is allowed to use. However, if μ is sufficiently high, eventually a proactive system will either fill all available capacity or place a copy of each object on every node.

This concern is not unique to proactive maintenance systems but proactive maintenance may cause the capacity limit to be reached sooner than in the reactive case. There are two general options for dealing with this situation: first, nodes that are at capacity could reject new replica creation requests until the application deletes some objects or additional local resources are made available. The alternative would be for each node to decide (perhaps in consultation with other nodes holding replicas) which objects are sufficiently highly replicated that it seems safe to delete its local copy.

Tempo attempts to skirt this problem by emulating the behavior of reactive replication systems: nodes agree to place a cap $R_{\max}$ on the total number of replicas that can be made of a given object. Tempo proactively produces replicas for objects up until this limit and then ceases to worry about any objects with more than that many replicas. The goal is to have enough copies to preserve durability but allow the system to reach its capacity limit due to too many objects rather than $R_{\max}$ copies of each object. The operators of a network of Tempo nodes might choose $R_{\max}$ such that they agree it is unlikely for that many nodes to simultaneously suffer disk failures.

3. IMPLEMENTING TEMPO

In this section, we discuss how to change an existing distributed hash table, DHash [6], to implement Tempo. For simplicity, we consider only redundancy obtained through replication though DHash supports erasure coded fragments. DHash uses consistent hashing to assign maintenance responsibility of each immutable object to its successor based on the cryptographic hash of the object itself (the content hash). When an object is inserted into DHash, the inserting node places r_L replicas on the first r_L successors of the object's key. The purpose of this redundancy is to protect against data loss due to a burst of $r_L - 1$ permanent failures that occurs before the objects on those nodes can be repaired. DHash uses a reactive repair system that repairs redundancy when the number of replicas of an object is found to be below r_L; each node is responsible for maintaining the redundancy level of the objects for which it is the successor. r_L is chosen to be less than the length of the successor list; in simulations.

Tempo also uses consistent hashing like DHash. Instead of monitoring for failures, each Tempo node has a bandwidth budget b_i that it uses to make additional replicas. Periodically, each node considers the objects for which it is responsible. Based on knowledge of where each object is replicated, objects with the lowest level of redundancy are chosen for replication on a random member of the successor list that is missing the object; in the event that the successor itself is missing the object, the object is downloaded locally. Replicas are placed on successors until each node on the successor list holds a replica. The level is redundancy is discovered via DHash's object synchronization mechanism which periodically exchanges information about which objects are stored on which nodes: the successor of each key maintains a database of this information on disk to avoid repeated exchanges of information. However, this is soft-state that can be recreated if the successor changes.

To maintain an average sending rate of no higher than b_i, Tempo uses a token bucket, much like Accordion [10]. A node is given a byte-credit periodically and spends that by making replicas. In particularly, every Δt seconds, the node's b_{avail} is incremented by $b_i \cdot \Delta t$; this accumulates until there is sufficient available bandwidth to initiate a repair. Once the repair is initiated, it actually proceeds at rate determined by the DHash congestion control protocol. To help balance the transfer workload, Tempo prefers downloading from its left and right neighbor in the Chord ring and downloads from each of them at half the bandwidth limit.

If all maintenance traffic were conducted in a special process or over a dedicated TCP/UDP port, one could achieve the same effect as the token bucket by using a traffic shaper (e.g. Trickle [7]) to regulate maintenance traffic. This would simplify the implementation further because it could simply create replicas all the time, without paying attention to bandwidth. However, the implementation would still need to balance the transfer workload to ensure that bandwidth budget of all nodes is utilized.

Handling of full disks remains an issue in our current design of Tempo. If new writes are blocked, some objects may eventually appear to be inaccessible due to the requirement that objects be held on nodes within the successor list. This is because growth in the system could move the original replica set out of the successor list while the current successor nodes are at capacity; without other book-keeping, this would give the appearance of some objects being unavailable (though they may still be durably stored). Though this is unlikely to occur, it may increase the appeal of deleting redundancy. A simple heuristic for this in successor placement would be to delete those objects for which the node is no longer in the successor list (and whose identifiers are furthest from the node). Such objects are being actively maintained within their current successor list, which suggests that it is safe to delete them.

4. IS PROACTIVE WORTHWHILE?

For Tempo to be viable, it must achieve durability comparable to state-of-the-art reactive systems while staying within its bandwidth budget. We evaluated Tempo in simulation against a trace of PlanetLab disk and transient failures. We used an event-driven simulator designed to capture the basic behavior of the different algorithms. Given a trace of node transient and permanent failures and a corresponding set of data objects, the simulator periodically records statistics such as the total bytes sent in the system, the number of live nodes, and the number of available objects.

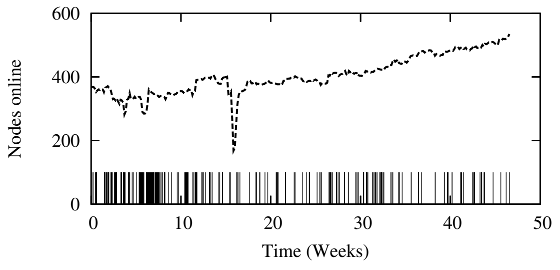

Figure 1: Number of online PlanetLab nodes and distribution of permanent failures.

Dates	24 Oct 2004 – 15 Sep 2005
Number of hosts	555
Number of transient failures	19127
Number of disk failures	628
Transient host downtime (s)	1232, 116451, 59427
Any failure interarrival (s)	162, 1481, 3898
Perm. failures interarrival (s)	832, 44990, 142336
(Median/Mean/90th percentile)	

Table 1: CoMon+PLC trace characteristics

4.1 Trace characteristics

In order to evaluate the behavior of Tempo, we constructed a trace that captures the relevant characteristics of PlanetLab over the past year: the rates of disk and transient failures. The trace is summarized in Table 1 and the number of available nodes over time is shown is shown in Figure 1. The data for this trace was obtained using CoMon data [12] and logs from PlanetLab Central [13].

CoMon has archival records collected on average every 5 minutes that include the uptime as reported by the system uptime counter on each node. We use resets of this system uptime counter to detect reboots and estimate the time when the node ceased being reachable based on the last time CoMon was able to successfully contact the node; this allows us to pinpoint failures without depending on the reachability of the node from the CoMon monitoring site.

We define a disk failure to be any permanent loss of disk contents, due to disk hardware failure or intentional or accidental erasing of the disk. In order to identify disk failures, the CoMon measurements were supplemented with event logs from PlanetLab Central. Each time a PlanetLab node is reinstalled (e.g., for an upgrade or after a disk is replaced following a failure), an event is logged. When we observe such an event, the node is considered offline until it is assigned a regular boot state.

The initial portion of the trace has an unusually high prevalence of permanent failures due to the deployment of PlanetLab V3. In the following experiments, we wait until the V3 rollout is essentially complete, insert 50,000 20MB objects with random keys and run the system for the remaining 10 months.

4.2 Results

To calibrate Tempo, we consider the bandwidth usage of Carbonite, a reactive algorithm for efficiently managing replicas in distributed systems [4]. Carbonite's main parameter is the repair threshold r_L; once the number of available replicas of an object falls below this threshold, it is repaired by Carbonite. With the setting $r_L = 3$, Carbonite uses on average approximately 1KBps of bandwidth per node, though it is highly bursty. To compare Tempo against Carbonite, we configured Tempo to initially insert 3 replicas as well, and we varied the bandwidth budget from 512 bytes per second to 4KB per second. In all cases, the length of the successor list is 15.

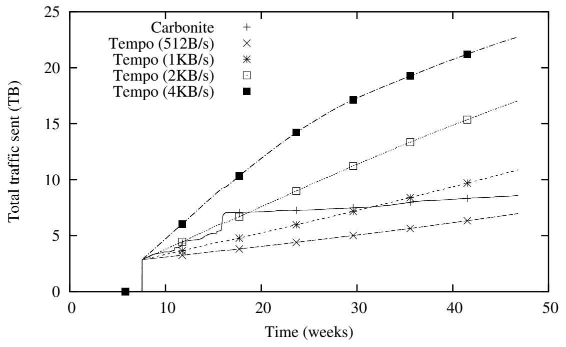

Figure 2: A comparison of cumulative bytes sent: the derivative shows instantaneous system bandwidth usage.

Figure 2 shows the cumulative number of bytes sent by Carbonite and Tempo. The derivative of each curve reveals the aggregate bandwidth used by each system. In the latter half of the trace, Carbonite uses almost no bandwidth, but there are periods where Carbonite uses orders of magnitude more aggregate bandwidth than Tempo. This can be seen at approximately week 15 of the trace when many nodes in PlanetLab underwent a rolling reboot—this resulted in a sharp drop and rise in the number of active nodes. During this period, Carbonite interpreted these transient failures as requiring repair and about 170 nodes transferred a total of nearly 2 terabytes of data over 2 days.

Figure 3 compares how the average bandwidth usage per node develops over time. As expected, nodes in Tempo use bandwidth very steadily, and with large budgets, the average bandwidth usage eventually declines as more and more objects are replicated across an entire successor list. Carbonite shows phases of high activity alternating with phases of almost no activity.

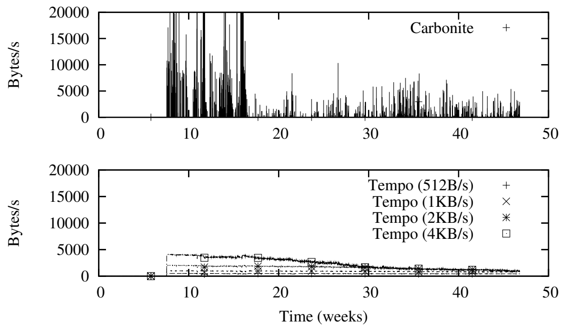

Figure 3: Average bandwidth usage per node.

Constantly creating additional replicas can result in a higher average number of replicas per object compared to simply responding to failures, as shown in Figure 4. However, Tempo can be tuned to not use significantly more bytes than Carbonite.

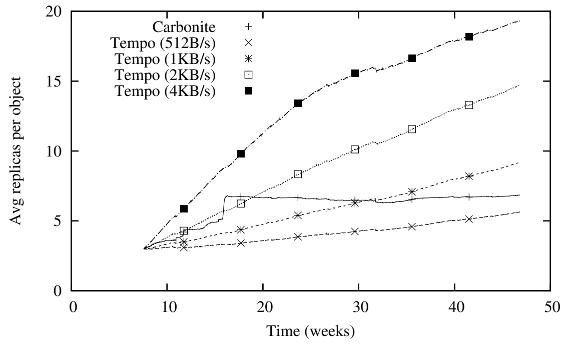

Figure 4: Average number of replicas per object.

Figure 5 shows the cumulative distribution of the number of replicas per object at the end of the trace. Over the course of the trace, even with only 512 bytes per second of replication bandwidth, Tempo is able to provide 99.8% durability: only 78 objects were lost. With higher bandwidth usage, both Tempo and Carbonite provide 100% durability over this trace. Both Carbonite and Tempo sometimes create 'extra' copies of an object because of transient failures. When the nodes with the original copy returns, these copies are not deleted, which is why the average number of replicas can be higher than $r_L = 3$ (for Carbonite) and higher than the successor list size of 15 (for Tempo).

As seen in the average node bandwidth usage, replicating at 4KBps is enough to reach the point where copies of an object are stored on every member of the successor list. When that happens, Tempo stops replicating that object further. The result of this is that doubling the budget available to Tempo from 2KBps to 4KBps does not create twice as many replicas.

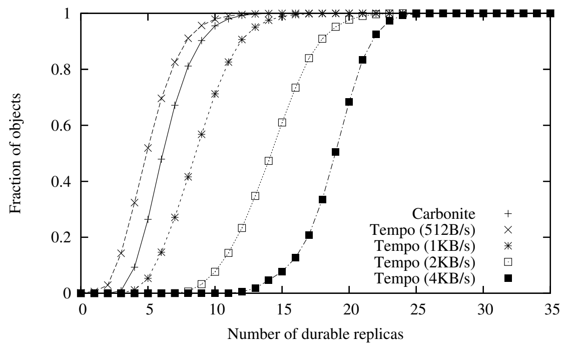

Figure 5: CDF of the number of replicas per object at the end of the experiment.

Tempo efficiently maintains replicas with successor list placement, as opposed to random placement (as exemplified by Total Recall [1]). While random placement improves the repair parallelism as described in [4], it makes the task of monitoring placement of objects more difficult: there is the concern that $O(N)$ monitoring traffic will become prohibitively expensive. By using otherwise idle bandwidth, Tempo allows for $O(\log N)$ monitoring traffic with successor list placement and provides comparable durability to a random placement system.

5. RELATED WORK

Most prior distributed hash tables monitor the availability of data and replace lost redundancy on other nodes in reaction to failures [6, 8, 18]. Total Recall proactively does work in bursts, creating up to a fixed number of additional erasure coded fragments in order to minimize bandwidth use and avoid responding to transient failures [1]. The goal of Total Recall is to achieve a fixed availability based on introspection; in contrast to Tempo, Total Recall does not limit the amount of bandwidth it uses.

In [4], we argue that creating replicas ahead of time is generally a disadvantage when the goal is to minimize bandwidth usage. Replicas created before the number of available replicas falls below the given threshold are either "extra" (would not have been needed) or would have been created eventually anyway. Tempo's goal is not to minimize total cost of maintenance necessarily but to keep the on-going cost of predictable. The analysis and models shown in [4] are similar to other contemporary work [14, 22].

Accordion [10] seeks to minimize routing latency subject to a bandwidth budget for table maintenance. The design of Accordion was driven by an analysis that showed that bandwidth was best used to learn through lookups and hide latency through parallel lookups. Tempo uses bandwidth to place replicas in suitable locations, and tries to avoid using bandwidth on exploration or dealing with failure.

Rhea *et al* observed that repairing routing tables in response to failure can be vulnerable to positive feedback loops. For example, a simple failure detector might measure network reachability. A packet loss rate may lead to a perceived failure and the corresponding increase in repair traffic could induce congestion and loss on additional links. This congestion could be interpreted as additional failures, leading to unpredictable and destructive behavior [17]. Similar observations were made regarding responding to transient failures in reactive replication systems [24].

Glacier [8] uses proactive repair, but with a different goal: While Tempo maximizes availability and durability given a bandwidth budget, Glacier maintains a durability goal under a set of failure assumptions, minimizing the required bandwidth. Once it reaches its target level of redundancy, it stops making replicas. Glacier also uses a bandwidth cap to limit the rate at which it generates repair traffic; this is to reduce the risk of congestion collapse when a large fraction of the replicas are destroyed in a large-scale correlated failure.

TCP Nice [23] is a modification to TCP's congestion control protocol to optimize it for background transfers like proactive replication: TCP Nice only transmits when there is no competition for bandwidth (e.g. no queueing) on a given path. Tempo could use TCP Nice for object replication.

Castro and Liskov presented a system that uses proactive recovery in the context of a Byzantine fault tolerant system [3]. In their system, a hardware watchdog periodically rebooted each server and reloaded software from secure storage regardless of whether a failure was detected. This helps minimize the potential window of time where each node could be faulty. This technique was extended to peer-to-peer distributed systems by Rodrigues *et al.* [20].

Other proactive replication systems are focused on performance. For example, Overcast [9] and Beehive [15] both pre-position replicas of data close to where it will be used before it is requested in order to minimize latency when a request is actually received.

6. CONCLUSIONS

Replica maintenance is critical to maintaining durability in wide area systems. We have proposed Tempo, a system that performs maintenance proactively instead of responding to failures. In contrast to systems that only create new replicas after a failure, Tempo constantly creates new replicas while operating within a user-specified bandwidth budget. This ensures that each node's bandwidth usage for maintenance is predictable, leaving most bandwidth free for actual application traffic. In simulations based on PlanetLab traces, Tempo is able to provide the same level of durability as traditional reactive systems using a comparable amount of bandwidth and without significant fluctuations in bandwidth usage. This result is encouraging enough that we plan to deploy and evaluate an implementation of Tempo on PlanetLab under real application workloads.

Acknowledgments The authors would like to thank Vivek Pai and Aaron Klingaman for their assistance in compiling the data used for the PlanetLab traces. This paper was also significantly improved as a result of feedback from the anonymous reviewers.

REFERENCES

[1] BHAGWAN, R., TATI, K., CHENG, Y.-C., SAVAGE, S., AND VOELKER, G. M. Total Recall: System support for automated availability management. In *Proc. of the 1st Symposium on Networked Systems Design and Implementation* (Mar. 2004).

[2] BLAKE, C., AND RODRIGUES, R. High availability, scalable storage, dynamic peer networks: Pick two. In *Proc. of the 9th HotOS* (Lihue, HI, May 2003), pp. 1–6.

[3] CASTRO, M., AND LISKOV, B. Proactive recovery in a byzantine-fault-tolerant system. In *Proc. of the 4th Symposium on Operating Systems Design and Implementation* (Oct. 2000).

[4] CHUN, B.-G., DABEK, F., HAEBERLEN, A., SIT, E., WEATHERSPOON, H., KAASHOEK, M. F., KUBIATOWICZ, J., AND MORRIS, R. Efficient replica maintenance for distributed storage systems. In *Proc. of the 3rd Symposium on Networked Systems Design and Implementation* (San Jose, CA, May 2006).

[5] DABEK, F. *A Distributed Hash Table*. PhD thesis, Massachusetts Institute of Technology, 2005.

[6] DABEK, F., LI, J., SIT, E., ROBERTSON, J., KAASHOEK, M. F., AND MORRIS, R. Designing a DHT for low latency and high throughput. In *Proc. of the 1st Symposium on Networked Systems Design and Implementation* (Mar. 2004).

[7] ERIKSEN, M. A. Trickle: A userland bandwidth shaper for unix-like systems. In *Proc. of the USENIX 2005 Annual Technical Conference, FREENIX Track* (Anaheim, CA, Apr. 2005).

[8] HAEBERLEN, A., MISLOVE, A., AND DRUSCHEL, P. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *Proc. of the 2nd Symposium on Networked Systems Design and Implementation* (May 2005).

[9] JANNOTTI, J., GIFFORD, D. K., JOHNSON, K. L., KAASHOEK, M. F., AND JAMES W. O'TOOLE, J. Overcast: Reliable multicasting with an overlay network. In *Proc. of the 4th Symposium on Operating Systems Design and Implementation* (Oct. 2000).

[10] LI, J., STRIBLING, J., MORRIS, R., AND KAASHOEK, M. F. Bandwidth-efficient management of DHT routing tables. In *Proc. of the 2nd Symposium on Networked Systems Design and Implementation* (May 2005).

[11] MISLOVE, A., POST, A., HAEBERLEN, A., AND DRUSCHEL, P. Experiences in building and operating a reliable peer-to-peer application. In *Proc. of the 1st EuroSys Conference* (April 2006).

[12] PARK, K. S., AND PAI, V. CoMon: a mostly-scalable monitoring system for PlanetLab. *ACM SIGOPS Operating Systems Review 40*, 1 (Jan. 2006), 65–74. `http://comon.cs.princeton.edu/`.

[13] PETERSON, L., ANDERSON, T., CULLER, D., AND ROSCOE, T. A blueprint for introducing disruptive technology into the Internet. In *Proc. of the 1st HotNets Workshop* (Oct. 2002). `http://www.planet-lab.org`.

[14] RAMABHADRAN, S., AND PASQUALE, J. Analysis of long-running replicated systems. In *Proc. of the 25th IEEE Annual Conference on Computer Communications (INFOCOM)* (Barcelona, Spain, Apr. 2006).

[15] RAMASUBRAMANIAN, V., AND SIRER, E. G. Beehive: O(1) lookup performance for power-law query distributions in peer-to-peer overlays. In *Proc. of the 1st Symposium on Networked Systems Design and Implementation* (Mar. 2004).

[16] RHEA, S., EATON, P., GEELS, D., SPOON, H. W., ZHAO, B., AND KUBIATOWICZ, J. Pond: the OceanStore prototype. In *Proc. of the 2nd USENIX Conference on File and Storage Technologies (FAST)* (Apr. 2003).

[17] RHEA, S., GEELS, D., ROSCOE, T., AND KUBIATOWICZ, J. Handling churn in a DHT. In *Proc. of the 2004 Usenix Annual Technical Conference* (June 2004).

[18] RHEA, S., GODFREY, B., KARP, B., KUBIATOWICZ, J., RATNASAMY, S., SHENKER, S., STOICA, I., AND YU, H. OpenDHT: A public DHT service and its uses. In *Proc. of the 2005 ACM SIGCOMM* (Philadelphia, PA, Aug. 2005).

[19] RODRIGUES, R., AND LISKOV, B. High availability in DHTs: Erasure coding vs. replication. In *Proc. of the 4th International Workshop on Peer-to-Peer Systems* (Feb. 2005).

[20] RODRIGUES, R., LISKOV, B., AND SHRIRA, L. The design of a robust peer-to-peer system. In *Proc. of the 10th ACM SIGOPS European Workshop* (St. Emilion, France, Sept. 2002).

[21] STRIBLING, J., COUNCILL, I. G., LI, J., RANS KAASHOEK, M. F., KARGER, D. R., MORRIS, R., AND SHENKER, S. OverCite: A cooperative digital research library. In *Proc. of the 4th International Workshop on Peer-to-Peer Systems* (Feb. 2005).

[22] TATI, K., AND VOELKER, G. M. On object maintenance in peer-to-peer systems. In *Proc. of the 5th International Workshop on Peer-to-Peer Systems* (Feb. 2006).

[23] VENKATARAMANI, A., KOKKU, R., AND DAHLIN, M. TCP Nice: a mechanism for background transfers. In *Proc. of the 5th Symposium on Operating Systems Design and Implementation* (Dec. 2002).

[24] WEATHERSPOON, H., CHUN, B.-G., SO, C. W., AND KUBIATOWICZ, J. Long-term data maintenance in wide-area storage systems: A quantitative approach. Tech. Rep. UCB//CSD-05-1404, U. C. Berkeley, July 2005.

[25] WEATHERSPOON, H., AND KUBIATOWICZ, J. D. Erasure coding vs. replication: A quantitative comparison. In *Proc. of the 1st International Workshop on Peer-to-Peer Systems* (Mar. 2002).