Efficient Replica Maintenance for Distributed Storage Systems

The area of focus of the Carbonite algorithm is durably and cost-efficiently storing of immutable objects in a system that aggregates the disks of many Internet nodes. Hence, Carbonite is a replication algorithm for large storage systems, tested in the paper on storing 1TB of data for a year. By focusing on durability, the paper provides a system able to maintain a greater amount of data than systems focusing on providing availability to the user. The algorithm works by first inserting a set of replicas for each object. Initially it creates new replicas in response to both transient (temporary) failures and permanent failures, but eventually it ends up almost exclusively creating replicas for permanent failures.

To ensure availability (immediate access to data), typical replication systems responds to disk failure by creating a new replica. When availability is key, replication is immediately used regardless if the failure was a disk failure or just a transient failure. The problem with this approach is that constantly making new replicas is bandwidth-costly, and do not scale to large systems. The developers of Carbonite realized that Internet users can tolerate some unavailability as long as they eventually will be able to view what they requested. Therefore they focused on making a durable system. They recognized that keeping data durable has three main challenges. First, keeping data durable, means that no data should be lost. This problem is usually solved by replicating the data as a response to disk failure. The problem with this approach is that network bandwidth is a scarce resource, meaning that constantly creating replicas in response to all types of failures is infeasible for an Internet scale system. The second challenge of durability is that to keep an object durable one actually only need to create replicas for disk failures, not for transient failures. However, there is no way of distinguish between the two, something that leads to systems having to create much more replicas than actually needed. The third challenge they discovered is that when systems do not perform reintegration of nodes that has suffered from a transient failure, the nodes are efficiently lost. These observations was the main motivation behind Carbonite.

As mentioned above, being able to maintain large storage application systems require sophisticated techniques for when and how replication is performed. Carbonite only makes replications in respond to failure if the number of replicas for a given object is below the threshold r_L, where r_L is the number of replicas needed to survive a burst of failures. This property is enforced by reintegration of failed nodes when they come back up again. By remembering which replicas was stored on a node, they can be reused, and thereby make sure that replication only is performed when the number of replications is below r_L. This technique drastically reduces the number of replications required. Since Carbonite creates new replicas when the number of available replicas is less than r_L it will have to keep track of more than r_L replicas. The number of replicas

needed is therefore dependent on the availability a, and shown in the paper to be approximately $\frac{2r_L}{a}$. Because of this, the paper discusses initially creating $\frac{2r_L}{a}$ replicas instead of r_L, but concludes that this approach is less attractive since it requires knowing the availability. Another technique to reduce the number of replications made as responds to transient failures is timeouts. By waiting a given amount of time before creating a new replica, the node might come back, and cancel the planned replication. However, for timeouts to be useful it is important that expected node lifetime is much longer than the timeout.

Carbonite is tested using the PlanetLab testbed and a synthetic trace, and the results are compared against Cates and TotalRecall as well as an oracle that knows if a failure is transient and thereby only creates replications in case of disk failure. For both test cases, Carbonite outperform Cates and TotalRecal, and is only beaten by the oracle; the perfect hypothetical system. Carbonite creates 44% more data than the oracle when keeping 1TB of data durable, while as Cates and TotalRecall uses almost a factor of two more network traffic than the oracle. Results also shows that as long as the target number of replicas r_L is well chosen, the system is durable. However, they do not have any results on availability which would be interesting to look at. This decision is justified by the authors' observation that users can tolerate some unavailability, but should be included because users will not tolerate too much unavailability either. The results shows that using timeouts leads to a decreasing amount of total bytes sent. However, as the timeout interval grows beyond a certain limit (varying for different values of r_L) durability falls drastically.

Carbonite can be applied to systems that require durably storing of large amounts of data. As they mention, the systems must be able to handle some unavailability. For instance, they mention that news paper readers can tolerate some unavailability as long as they are able to read the news article at some point. The authors also discusses that Carbonite can be applied for implementing distributed hash tables, and directory-style systems.

Carbonite is compared against different systems and the results in the paper looks promising when it comes to durability in a large scale system. However, as mentioned above, the paper does not present any results on the availability of the system. Even though durability might be a more practical and useful goal than availability for this kind of system, it will be useless if none of the objects are ever available. While the research presented is promising it is yet to be tested on a real-world application. A few assumptions, that might not be that realistic, such as assuming that the disks fail independently, no shared bottlenecks, unlimited disk capacity for nodes, and that available nodes are reachable from all other nodes, should be tested. The good thing is that the authors acknowledge the need for real-world testing and have developed prototypes for different applications for which they promise to report on long-term experience with.