Walking toward moving goalposts: agile management for evolving systems

Richard A. Golding and Theodore M. Wong
IBM Almaden Research Center, San Jose, CA

Abstract

Much of the practical work in the autonomic management of storage systems has taken the "bolt-on" approach: take existing systems and add a separate management system on the side. While this approach can improve legacy systems, it has several problems, including scaling to heterogeneous and large systems and maintaining consistency between the system and the management model. We argue for a different approach, where autonomic management is woven throughout a system, as in the K2 distributed storage system that we are implementing. This distributes responsibility for management operations over all nodes according to ability and security, and stores management state as part of the entities being managed. Decision algorithms set general configuration goals and then let many system components work in parallel to move toward the goals.

1. Introduction

Most storage system management applications are built as bolt-on additions to existing systems. For example, IBM TPC [9], EMC ControlCenter [5], and HP Labs Minerva [2] all provide management for existing storage systems. All these applications have improved the manageability of existing systems, and have introduced elements of autonomic management. They generally work by building a model of the system in an external application, applying monitoring information to that model, and using the model for analyzing and planning how to react to events in the system (the MAPE loop).

The model-based, bolt-on approach is not working well in these products, even though they are on the market and providing useful tools. The deficiencies arise from centralization and separation. The systems cited do not scale with the size of the system being managed; instead, they build a single centralized model of the entire system, and run decision-making algorithms inside the management application. Further, separating the management application from the system allows inconsistency between the two, and makes the system vulnerable to management application failure as well as to internal failure. A separate management application has longer latency for responding to system events than mechanisms built into the system. While management applications built in a more distributed fashion are possible, most systems on the market are not structured that way.

We are exploring an alternative approach in K2, the distributed storage system that we are building. K2 has no central authority that defines the system. Instead, it is structured as a federation of semi-independent systems, which we expect to lead to a robust system that can adapt quickly to changes in requirements or environment. This model has worked well for existing Internet services, including the Web, DNS, and some grid applications, where different pieces of the service are provided by different organizations.

As in biological nervous systems, we weave autonomic functions throughout the system with many components working in parallel to keep the system going. This approach scales the resources used for management with the size of the system being managed. By integrating management into the system, there is no possibility of a model becoming inconsistent or failing separately. Integration also ensures that components close to a problem detect and resolve problems quickly.

The self-management mechanisms in the K2 prototype are concerned only with resource allocation, which provides a clean interface on which to layer higher-level application and user management. The system provides different users with separate *pools*, each of which has resource requirements. The system treats resource allocation as a constrained optimization problem, working to ensure that each pool is provisioned to meet the requirements, and that load is balanced between storage servers.

K2 focuses on an incremental, online solution in which each pool is managed separately as much as possible. A pool is given a feasible resource assignment when first created, and then the system incrementally improves the assignment over time as more information becomes available about actual resource usage. This approach is similar to many optimization heuristics that start with an easily-

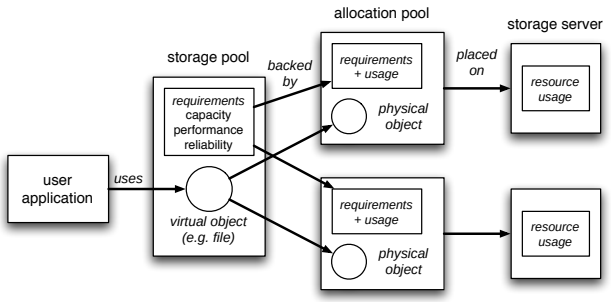

Figure 1. Components in the K2 storage pool mechanism.

computed feasible solution and then incrementally "hill-climb" toward better solutions. In those circumstances where one pool cannot be managed by itself – such as recovering from a failure that affects multiple pools – the affected pools coordinate to make a more global decision.

In this paper we present some of the key components of the K2 prototype that map user requirements onto specific resource allocations, decide how to react to changes in requirements or the storage server population, and migrate data to effect the decisions.

2. System model

K2 implements distributed storage over a cluster of small, self-contained storage servers that presents an abstraction of *storage pools* containing *virtual objects* to higher-level applications (Figure 1). Storage pools are virtual containers with a set of capacity (in bytes), performance (in I/O rate), and reliability (in mean time to data loss) requirements. Capacity and performance requirements represent resources reserved for the pool. Reliability determines the redundancy codes acceptable for storing data in the pool. Virtual objects are byte containers of variable length, somewhat in the style of the SCSI OSD storage model [10].

A storage pool is backed by a set of physical containers called *allocation pools* (APs) on a set of storage servers. Like a storage pool, an AP specifies requirements on capacity and performance resources; the aggregate of a set of AP requirements equals their parent pool's requirement. K2 assigns each AP in a pool to a different server, but may assign APs of multiple pools to a single server.

A virtual object is backed by a set of *physical objects* in a set of APs, analogous to how storage pools are backed on storage servers by APs. Data for one virtual object is spread across its physical objects according to a redundancy code determined by the storage pool's reliability requirement.

Each server in a K2 cluster contains a CPU and disks. We expect that the population of servers in an operational K2 cluster will be large (tens of thousands), heterogeneous (servers will have different CPUs or disk capacities), and unstable (additions, removals, and failures of servers will be relatively frequent). Each server knows its capacity and performance resource availability.

K2 provides a reliable platform for autonomic management by electing a *manager* for each storage pool, which monitors and analyzes the state of the pool and coordinates responses to state changes. The APs in a pool execute a leader-follower election protocol [7] to spawn a pool manager on some server. Once elected, the manager monitors the APs for failures or resource usage changes; when detected, the manager runs a decision algorithm (§3) to determine the response. Each AP maintains a persistent copy of the federation state (such as membership, AP to storage server allocation bindings, and virtual object layout metadata), while the manager may cache a transient copy. After making a configuration decision, the manager pushes the change out consistently to the APs using a timestamp-based transaction protocol [3]. If physical object data needs to migrated from one AP to another, the manager uses the transaction protocol to read and write the objects. The transaction protocol ensures that the migration can proceed in parallel with client accesses.

By giving each pool its own manager, K2 sidesteps many of the scalability problems that arise when using centralized managers to provide equivalent services. In particular, the effects of the failure of a storage server only propagate through to any hosted APs and their parent pools; other APs and pools generally remain unaffected.

The AP requirements act as a contract between manager and managed, simplifying the pool manager by delegating the details to each AP. Each storage server manages capacity and performance resource usage to ensure that it meets the requirements of all hosted APs, and that I/O streams to each AP do not interfere with each other. In particular, it shapes the I/O traffic [15] to fulfill the performance reserves of all APs, and shares any remaining performance resources fairly among APs.

Finally, K2 creates two special storage pools for internal purposes. The *spare reserve* sets aside sufficient resources to cover expected failures – typically a few storage servers' worth. Since we expect that failure and imbalance are normal conditions in a large system, the *management reserve* sets aside a fraction of each server's performance resources for use when migrating or recovering data (§5). Each of these special pools is represented by a set of APs and elects a manager, just like other pools.

This model for distributed storage is similar to other systems that use virtualized objects, including the UCSC Swift file system [4], the Panasas ActiveScale file system [12], and the CMU Self-*/Ursa Minor system [1, 6].

K2 is closest in spirit to Ursa Minor in that we are both exploring self-management that is integrated into a storage system's design, and in particular both set up management for a set of workers (APs). There are three major differences. First, the Self-* project aims higher than K2: it potentially includes all aspects of system tuning, while K2 is focused just on resource provisioning given a set of requirements. The resource management interface in a K2 storage pool could fit into the Self-* vision as the language that a second-level manager uses to give instructions to a first-level manager. Second, while the Self-* vision includes a hierarchy of managers, the Ursa Minor prototype to date does not yet implement the vision of hierarchical management and instead has focused on implementing versatile, reliable object storage. In contrast, the K2 implementation to date focuses more on management and less on the actual storage. Finally, there is a difference in the philosophy behind the designs. Self-* and Ursa Minor are designed around a top-down system model, where the system is defined by a single hierarchy of managers and workers. This philosophy tends to produce a system familiar to those who expect an appliance, which is well-suited to use by a single organization. K2, on the other hand, is designed around a bottom-up philosophy, where the "system" is defined by a federation of many semi-independent systems in the form of pools and servers. They are likely owned and used by many different users, and form collective decision-making mechanisms at need. This approach produces a system in the style of common Internet services such as the Web or DNS, which are intended for use by a large, diverse community.

3. Decision algorithms

A pool's manager is responsible for computing the resource configuration for the pool's APs. The configuration must respect the capacity, performance, and reliability requirements on the storage pool, and should try to balance usage across APs and unused resources across storage servers. A new pool is configured based on its declared requirements; from time to time, the manager recomputes the configuration based on actual usage to try to balance usage. We chose to measure balance as the variance in unused resource in the system, with low variance being better. When rebalancing, the manager will only use the new configuration if it decreases this variance – that is, if it moves the total system toward a better configuration.

In K2, we formulate the configuration decision process as a constrained (bin-packing) optimization problem, as others have done [2], but focus on online, incremental heuristics. We have developed the *MinDot* algorithm, which is based on the vector dot product model from the Toyoda heuristic [14]. MinDot is an online algorithm, where a pool manager sends a vector representing total required resources to the storage servers it can use, and each server returns the dot product of the requirements with its available resources. The dot product can be treated as the server's bid for storing part of the pool. MinDot determines what fraction of the storage pool should be assigned to each storage server in order to minimize the sum of dot products across all used servers (hence the name of the algorithm), subject to constraints. This can be done as a linear, greedy assignment. This approach tends to bias usage toward the servers that have the most unused resources and balances usage across multiple dimensions (such as capacity and performance).

The pool's reliability requirements generate the most constraints. A particular level of reliability requires at least n-way redundancy, and so no more than $1/n$ of any resource can go on any one server; similarly, there is an upper bound on the number of servers that can be involved before reliability drops below the required level. Using an analytic model, the manager translates the MTTDL requirement into these bounds and the kind of redundancy code to be used [13]. The model currently does not cope with heterogeneous servers and changing estimates of server reliability, which are topics for future work.

This approach meets our goals of independent and efficient decision-making: each manager can find available resources on its own. In a large system, many managers will be working in parallel to improve system configuration. If they make concurrent, conflicting decisions, one or more managers will fail to allocate the resources they want, and will retry their decision-making taking the changed resource availability into account. This approach is simple but not formally live; we are looking into more robust mechanisms, perhaps based on the mechanism we use in manager election.

There are several other areas for future improvement. While the online bin-packing algorithm appears to work well in practice, we do not have a formal analysis of how far the on-line variant of the heuristic is from batch heuristics. The algorithm we describe also ignores complex workload specifications; for example, correlation between workloads or workload behaviors such as sequentiality. Our experience is that users are rarely able to specify these behaviors manually, and so we are hoping that other research projects will generate tools that learn such facts from observation, after which we expect to incorporate them into our decision algorithm.

There is one significant exception to the decision-making approach just discussed: there are failure cases where each pool manager cannot act independently. Instead, multiple pools must jointly compute a sequence of pool reconfigurations that will restore redundancy. The spare reserve (§2) ensures that there will be enough resources in the system in

general to handle some number of storage server failures; however, sometimes the spare resources may not be on the right servers. For example, some pool that has suffered a failure may have its APs on the same servers as all the spare reserve, and so to meet reliability constraints spare space must be generated on some other server by migrating part of some other pool before the damaged pool can find acceptable spares and get rebuilt. We believe that this becomes less likely as the size of the system increases. We have currently implemented a simple backtracking search algorithm to find a feasible sequence of pool reconfigurations. This is another area for future research.

4. Configuration goals

While the decision algorithm can quickly make a decision about how to reconfigure storage pools, it takes time to react to the decision by migrating data, and conditions may change while the system is moving toward the desired configuration. This is especially complicated when multiple pools must be reconfigured in response to failure. Many existing systems use a simple interpretation of the MAPE model, generating a specific plan of the steps required to move the system to a new configuration. The plan becomes invalid when the system needs to move to a different configuration.

K2 takes an alternate, more fluid path, by separating decision and execution. The decision algorithms determine the desired future configuration, and label each AP with its *goal* configuration – to shrink or grow each resource, or to go away – representing where the system should go, but not how to get there. The system then moves toward those goals by incrementally migrating or rebuilding virtual object data in many small steps. The decision algorithms at the AP level generate plans that do not deadlock, after which the migration can reach the intended configuration incrementally.

If conditions change part way through reconfiguration, the decision algorithms place new goals on the APs and the system starts moving in that new direction. No explicit migration plan is required in advance.

The goals are stored as a second set of resource attributes on the AP, in parallel with the attributes that reflect current actual resource assignment. This approach integrates the plan (the goals) with the AP and makes the goals persistent, so that the decisions survive manager failure, and there is no possibility of inconsistency between manager and AP, reducing implementation complexity.

The execution is further separated between intra- and inter-server aspects. The migration and recovery mechanisms in §5 determine what data should move between APs. Within a storage server, APs with growth goals are given resources when they become available, and when resources are freed up in an AP with a shrinkage goal, the resources are given to some other AP. In this way distributed mechanisms are not needed to track detailed changes to resource allocation, improving execution efficiency and implementation complexity.

5. Migration and recovery

Because the managers express their reconfiguration decisions as coarse-grained goals on APs, the migration and recovery mechanism must move actual resource usage toward those goals by incrementally moving one virtual object after another. The mechanism must do so as quickly as possible while handling the possibility that one pool may not be able to make progress until another pool has moved some of its data out of the way.

Migration and recovery work as follows. Whenever a pool manager has some AP with a goal or has objects with degraded redundancy, it picks one object in the pool for migration to move the APs toward their resource goals. The manager computes a new placement for the object, and moves data appropriately. The new object placement uses placement rules that avoid APs with shrinkage goals and favor APs with more available resources. (While our current prototype migrates or recovers one object at a time, a production system would process several objects in parallel to improve throughput.)

First priority goes to recovering degraded objects, because restoring redundancy quickly is important to overall system reliability. Second priority is for objects that are partially stored on any AP has a goal of shrinking, because migrating that object will open up resource for some other AP in some other pool. Last priority goes to any other object, since migrating it might improve resource balance. In some cases no object can be migrated until some other pool has moved data out of the way.

The pool migration mechanisms have the potential to interfere with application traffic. K2 addresses this in two ways. First, the microtransaction protocol that K2 clients use for accessing data ensures that client accesses can proceed in parallel with migration without compromising consistency or performance [3]. Second, the system sets aside a management reserve, a fraction of the performance resources of every node, and each node runs an I/O scheduler that ensures that migration traffic uses the reserved resources when clients are concurrently performing normal I/Os using the APs' reserved resources.

This approach is different from other systems, which more often have developed fine-grained migration plans rather than moving incrementally. For example, the migration planning work by Hall *et al.* [8] builds a plan for the entire data migration at once, based on complete knowledge of all data placement changes to be done. The plan completes in as few steps as possible, while respecting resource

constraints on all nodes. However, the efficiency of the plan comes at a cost of running an expensive planning algorithm, and having to start the planning from scratch if the desired new configuration changes. Other migration planners based on job-shop scheduling [11] have similar properties. In contrast, the K2 mechanisms compute only a coarse-grained plan – at the level of APs – after which the object-by-object migration can proceed incrementally, knowing that there is some feasible sequence of object migrations that will effect the plan. The K2 mechanisms might not find as short a sequence of moves to complete the migration, but they avoid wasted detailed planning if the system environment changes during migration, requiring re-planning.

6. Conclusions

In summary, K2 provides autonomic resource management by 1) providing failure-tolerant pool managers as a management platform; 2) having each manager monitor the state of its pool and set coarse-grain incremental configuration goals by running decision algorithms; and 3) using migration and recovery mechanisms that incrementally move the system toward those goals. The decision algorithms work to ensure full provisioning for each pool, while balancing resource usage across storage servers. If conditions change, the managers will change goals and migration will begin moving toward the new goals.

The K2 approach to autonomic resource management differs from other work in two ways. It is built around a model that avoids a central authority, and instead elects management services as needed. The resulting decentralization allows parallel management operations in most cases. Additionally, while K2 can be thought of as using a MAPE loop, it avoids sharp divisions between the steps. Instead, it works fluidly by continuously developing new goals and moving the system toward a better state in many small steps, which makes the system responsive to change.

We have prototyped these mechanisms, and found them to work well in the prototype. There remain several interesting research questions to investigate as we refine the prototype and test it at increasingly larger scale.

7. Acknowledgements

We thank Darrell Long, UC Santa Cruz, for helpful comments during the writing, and the anonymous reviewers, for their feedback that helped us clarify the presentation.

References

[1] M. Abd-El-Malek, W. V. Courtright II, C. Cranor, G. R. Ganger, J. Hendricks, A. J. Klosterman, M. Mesnier, M. Prasad, B. Salmon, R. R. Sambasivan, S. Sinnamohideen, J. D. Strunk, E. Thereska, M. Wachs, and J. J. Wylie. Ursa Minor: Versatile cluster-based storage. In *Proc. of the 4th Conf. on File and Storage Technology*. USENIX Assoc., Dec. 2005.

[2] G. A. Alvarez, E. Borowsky, S. Go, T. H. Romer, R. Becker-Szendy, R. Golding, A. Merchant, M. Spasojevic, A. Veitch, and J. Wilkes. Minerva: An automated resource provisioning tool for large-scale storage systems. *ACM Trans. on Comp. Sys.*, 19(4):483–518, Nov. 2001.

[3] K. Amiri, G. A. Gibson, and R. Golding. Highly concurrent shared storage. In *Proc. of the 20th Intl. Conf. on Distributed Computing Systems*, pages 298–307. IEEE, Apr. 2000.

[4] L.-F. Cabrera and D. D. E. Long. Swift: Using distributed disk striping to provide high I/O data rates. *Computing Systems*, 4(4):405–436, 1991.

[5] EMC. EMC ControlCenter software family data sheet. http://www.emc.com/products/storage_management/controlcenter/pdf/H1082_CC_Stor_Fam_LDV.pdf, 2004.

[6] G. Ganger, J. Strunk, and A. Klosterman. Self-* storage: Brick-based storage with automated administration. Tech. Report CMU-CS-03-178, Sch. of Computer Science, Carnegie Mellon University, Pittsburgh, PA, Aug. 2003.

[7] R. Golding and E. Borowsky. Fault-tolerant replication management in large-scale distributed storage systems. In *Proc. of the 18th Symp. on Reliable Distributed Systems*, pages 144–155. IEEE, Oct. 1999.

[8] J. Hall, J. D. Hartline, A. R. Karlin, J. Saia, and J. Wilkes. On algorithms for efficient data migration. In *Proc. of 12th Ann. Symp. on Discrete Algorithms*, pages 620–629, 2001.

[9] IBM Corp. IBM TotalStorage Productivity Center. http://www.ibm.com/servers/storage/software/center/index.html, 2004.

[10] INCITS Technical Committee. Information technology - SCSI object-based storage device commands - 2 (OSD-2). http://www.t10.org/ftp/t10/drafts/osd2/osd2r00.pdf.

[11] J. Y.-T. Leung, editor. *Handbook of scheduling: Algorithms, models, and performance analysis*. Chapman & Hall/CRC, 2004.

[12] D. Nagle, D. Serenyi, and A. Matthews. The Panasas ActiveScale storage cluster—Delivering scalable high bandwidth storage. In *Proc. of the 2004 ACM/IEEE Conf. on Supercomputing*, Nov. 2004.

[13] K. Rao, J. L. Hafner, and R. A. Golding. Reliability for networked storage nodes. In *Proc. of DSN 2006, the Intl. Conf on Dependable Systems and Networks*, 2006.

[14] Y. Toyoda. A simplified algorithm for obtaining approximate solutions to zero-one programming problems. *Management Science*, 21(12):1417–1427, Aug. 1975.

[15] T. M. Wong, R. A. Golding, C. Lin, and R. A. Becker-Szendy. Zygaria: Storage performance as a managed resource. In *Proc. of the 12th IEEE Real-Time and Embedded Technology and Applications Symp.*, Apr. 2006.