Data-Centric Reconfiguration with Network-Attached Disks

Alexander Shraer*
Dept. of Electrical Engineering
Technion, Haifa, Israel
shralex@tx.technion.ac.il

Jean-Philippe Martin
Microsoft Research
Mountain View, CA, USA
jpmartin@microsoft.com

Dahlia Malkhi
Microsoft Research
Mountain View, CA, USA
dalia@microsoft.com

Idit Keidar
Dept. of Electrical Engineering
Technion, Haifa, Israel
idish@ee.technion.ac.il

ABSTRACT

We consider data-centric distributed storage, where storage-nodes are directly attached to the network. We present DynaDisk, the first read/write storage system that allows clients to add and remove storage devices in a completely decentralized manner, and without stopping ongoing read/write operations. DynaDisk supports two alternative approaches to reconfiguration, one partially synchronous (consensus-based) and one asynchronous. We evaluate DynaDisk on a LAN cluster and compare these two reconfiguration methods.

Categories and Subject Descriptors

B.3.2 [**Memory Structures**]: Design Styles—*shared memory*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications*; D.4.2 [**Operating Systems**]: Storage Management—*secondary storage, distributed memories*; D.4.5 [**Operating Systems**]: Reliability—*fault-tolerance*; H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*distributed systems*

General Terms

Algorithms, Design, Reliability, Performance

Keywords

Shared-memory emulations, dynamic systems, atomic storage.

1. INTRODUCTION

Distributed storage architectures [10, 16, 1] provide a cheap and scalable alternative to expensive monolithic disk array systems currently used in enterprise environments. Such distributed architectures make use of many unreliable *storage-nodes* directly attached to the network and provide reliability through redundancy, e.g., by replicating each object on 3 nodes. We focus on large scale distributed storage that provides read/write functionality with atomic guarantees.

The large number of fault-prone storage-nodes requires supporting dynamic configuration changes when faulty nodes are removed and new ones are introduced. When such reconfigurations occur, proper coordination is essential to avoid "split-brain" behavior. In this paper we consider the problem of reconfiguration in the *data-centric* system model [8], motivated in Section 1.1, where storage-nodes are accessed directly through the network by many ephemeral clients. Neither the storage devices nor the clients communicate with their peers.

We present DynaDisk, the first data-centric read/write storage system that reconfigures in a completely decentralized fashion. DynaDisk can be configured to either use an asynchronous or a partially synchronous (consensus-based) algorithm for reconfigurations. In fact, one of our main goals is to compare the two reconfiguration approaches. Our solution can be seen as a data-centric version of our DynaStore algorithm [3] (see Section 1.2). DynaDisk encapsulates the core mechanism enabling reconfigurations, (whether asynchronous or partially-synchronous), in an object with similar guarantees to the *weak snapshots* of [3]. Thus, DynaDisk provides a unified framework for evaluating different reconfiguration approaches. In particular, in order to compare the consensus-based reconfiguration approach to an asynchronous one, we implement weak snapshots using each approach. We implement weak snapshots in the asynchronous data-centric model for the first time (Section 3). Note that the storage overhead of message-passing snapshot algorithms [2, 3] is linear in the number of coordinating parties. Thus, if adopted naïvely to the data-centric model, the overhead would be linear in the number of clients, which is prohibitive as it can be unbounded. Our new asynchronous implementation is conceptually different and entails an overhead proportional only to the number of storage-nodes currently in the system. Beyond weak snapshots, DynaDisk makes several other modifications to adapt DynaStore to our model (see Section 4). These include eliminating the broadcast of new configurations among storage-nodes, adding support for multiple objects (DynaStore implements a single object), and incremental state transfer.

*Supported by Eshkol Fellowship from the Israeli Ministry of Science.

We evaluate DynaDisk to investigate an interesting question – what coordination mechanism is preferable in practice? Our evaluation (see Section 5) shows that, compared to the consensus-based approach, asynchronous reconfigurations have a significant negative effect on latency of read and write operations that execute concurrently with reconfigurations. Essentially, this inherently stems from the fact that a consensus-free algorithm must sometimes work with multiple configurations that it considers possible, whereas an algorithm based on consensus can always work with a single configuration on which all clients agree. Having said that, the asynchronous algorithm achieves a slightly better and much more predictable reconfiguration latency when many reconfigurations occur simultaneously. In such extreme situations, the consensus-based algorithm sometimes takes a long time to reach a decision (in theory, reconfigurations are not guaranteed to complete [9]).

1.1 Why data-centric?

We believe that in distributed storage, replication algorithms should be separate from replica state. This is enabled by the data-centric approach [8] which enforces a two-tier architecture, where replication protocols are executed by clients, and storage-nodes simply provide persistent storage. The alternative approach that requires communication among storage-nodes is undesirable for many reasons. First, it unnecessarily complicates storage functionality. In fact, even though advances in storage technology allow customization of controller logic [11], application-dependent communication among storage-nodes would turn storage-nodes into servers with local disks. Allowing clients to directly access the storage simplifies control logic and reduces the number of fault-prone system components. In addition, enabling the disks to transfer data directly to clients eliminates the server bandwidth bottleneck.

Whether disks are directly connected to the network or through servers, the data-centric approach increases system throughput as storage-nodes are able to respond to clients immediately, without prior coordination involving other nodes.

Finally, note that each object is usually stored on a different small subset of storage-nodes and each storage-node hosts a vast amount of objects. Therefore, in a non-data-centric solution, each storage-node has to communicate with a large overall number of peers coordinating updates on all of its objects. As TCP is the transfer protocol most used in datacenters [5], this may lead to a scalability problem since the number of connections is expected to be very large.

1.2 Related work

Data-centric read/write storage is considered in many works, e.g., [1, 6, 14, 15]. Most of these, however, assume a static world, where the set of storage devices is fixed from the outset. The only exceptions we are aware of are Ursa Minor [1] and the work of Martin et al. [15], which employ a centralized sequencer for configuration changes. Unlike Ursa Minor [1], the protocol of Martin et al. [15] allows read/write operations to continue during reconfigurations. DynaDisk is completely decentralized – it allows every client to reconfigure the system without communicating with other clients, and also allows read/write operations to continue.

Reconfiguration was mainly considered in models where servers storing object replicas communicate. This enabled solutions where servers run consensus or virtual synchrony algorithms to agree on the configuration [13, 16]. Recently, Aguilera et al. [3] proposed DynaStore, a reconfigurable read/write storage algorithm based on consensus-free coordination also using direct communication among storage servers. In the data-centric model, however, the reconfiguration problem is more challenging since storage-nodes cannot coordinate directly, whereas clients are mostly disconnected from the system.

2. SYSTEM ARCHITECTURE

We assume an unknown, unbounded universe of storage-nodes Π and infinitely many clients that access them over the network. All clients and storage-nodes can crash, though as we mention below, excessive storage-node crashes can hamper liveness. The system emulates multiple atomic (high-level) objects with a *read* and *write* interface, provided as a client-side library. Each high-level object is replicated, and is stored in multiple basic storage units called *base-objects*, each residing on one of the storage-nodes. The initial value of every high-level object is $\perp$ and *base-objects* are created only when a non-$\perp$ value is written to the high-level object for the first time.

In addition to *read* and *write* operations, clients expose a *reconfig* interface, which allows for adding and removing storage-nodes and returns the new configuration. We say that a storage-node i is *active* if i does not crash, some client invokes a *reconfig* operation to add i, and no client invokes a *reconfig* operation to remove i. We assume that each active storage-node i responds to client requests starting from the time a *reconfig* operation adding i is invoked.

Obviously, system liveness depends on *reconfig* operations – for example, if all storage-nodes are removed, liveness is inevitably lost. Intuitively, each *read*, *write* and *reconfig* operation is guaranteed to complete as long as changes to the system are introduced gradually, and there is only a finite number of changes that occur during the operation (a formal liveness definition appears in [3]). A prerequisite for operation liveness is that clients are able to find a current set of storage-nodes. We discuss this further below.

Storage-nodes' interface. Clients can invoke $read_i$ and $write_i$ operations on every storage-node $i \in \Pi$. These operations can operate on one or more base-objects stored on i. Multi-object operations only access individual base-objects atomically – they are simply an optimization and can be replaced with multiple single-object operations.

We further assume that the storage-nodes support *read-modify-write* access, e.g., a *compare&swap* operation, which updates an object only if it equals to some value and returns the old value. This simple additional functionality does not require full customization of controller logic, as in Active Disks [11], and can be reused by multiple applications. Note that it is impossible to use a collection of fail-prone read-modify-write objects to emulate a reliable one [12] or solve consensus, but they can be used to enable reliable atomic read/write storage [4].

Algorithm 1 Asynchronous weak snapshot algorithm.

1: **Base-objects:** $\forall i, j \in S$, $N_i[j]$ is a base-object on storage-node i, initially $\perp$

2: **operation** *update*(c)
3: $(j, c') \leftarrow (\perp, \perp)$
4: $S' \leftarrow$ any subset of size $\lceil (|S|+1)/2 \rceil$ from S
5: invoke in parallel for all $i \in S'$:
6: $prev \leftarrow compare\&swap_i(N_i[i], \perp, c)$
7: **if** $prev \neq \perp$ **then** $(j, c') \leftarrow (i, prev)$
8: **else** $(j, c') \leftarrow (i, c)$
9: **wait** until $(j, c') \neq (\perp, \perp)$
10: $M \leftarrow \emptyset$
11: invoke in parallel for all $i \in S$:
12: $M \leftarrow M \cup \{write_i(N_i[j], c')\}$
13: **wait** until $|M| \geq \lceil (|S|+1)/2 \rceil$
14: **return** OK

15: **operation** *scan*()
16: **if** *collect*() $= \emptyset$ **then return** $\emptyset$
17: **return** *collect*()
18: **procedure** *collect*()
19: $M_1, M_2 \leftarrow \perp$; $L \leftarrow \perp^{|S|}$
20: invoke in parallel for all $i \in S$:
21: $M_1 \leftarrow M_1 \cup \{read_i(N_i)\}$
22: **wait** until $|M_1| \geq \lceil (|S|+1)/2 \rceil$
23: **forall** $i \in S$ s.t. $L'[i] \neq \perp$ for some $L' \in M_1$
24: $L[i] \leftarrow L'[i]$
25: $I \leftarrow \{i \mid L[i] \neq \perp\}$
26: invoke in parallel for all $i \in S$:
27: $M_2 \leftarrow M_2 \cup \{write_i(N_i\{I\}, L\{I\})\}$
28: **wait** until $|M_2| \geq \lceil (|S|+1)/2 \rceil$
29: **return** $\{L[i] \mid L[i] \neq \perp\}$

Service discovery. Finding the service is usually overlooked or not treated explicitly in theoretic distributed computing literature dealing with dynamic services. To address this issue, we model the service discovery component as a publicly known *oracle*, accessible by all clients. When queried by a client, the oracle returns some configuration previously returned by a *reconfig* operation, or the initial configuration if no *reconfig* has yet completed. If reconfigurations stop and the oracle is queried infinitely many times, it is assumed to eventually output the last configuration of the system.

The oracle can be implemented in different ways, e.g., using a publicly known naming service, such as DNS or UDDI. In case all storage-nodes are located inside an organization's network, it can be implemented by having clients broadcast a discovery request (e.g., as in ARP or Web Services Dynamic Discovery protocol). Note that after initially finding some node in the system, clients can usually proceed without the oracle, and may learn the latest configuration by contacting a storage-node they already know. Nevertheless, if the system reconfigures too quickly, they might need to contact the oracle again. It is important to note that the functionality we require of the oracle is very weak and cannot be used, e.g., to reach consensus in an asynchronous model. Finding the weakest sufficient oracle is an interesting direction for future work.

3. DISTRIBUTED WEAK SNAPSHOTS

A *weak snapshot* distributed object (WS, introduced in [3]) exposes two operations: *update(c)* and *scan()*. Weak snapshot is implemented on some fixed set S of storage-nodes and can be accessed by any number of clients. A client calls *update(c)* to propose a new value c, and calls *scan()* to retrieve a subset of previously proposed values. When used for reconfigurations in DynaDisk, c is a set of proposed configuration changes.

WS ensures the following safety guarantees [3]: Once at least one *update* completes, every newly invoked *scan* returns a non-empty set of updates; and once some value is returned in one complete *scan*, it is returned by all *scan*s invoked thereafter. Moreover, WS ensures that there exists some common update value c that is returned in all non-empty *scan*s. Intuitively, c can be seen as the first value to have been proposed, although in the presence of concurrent *update*s there isn't always a clear notion of the first one. WS ensures liveness of all operations provided that a majority (or quorum) of the fixed set of storage-nodes holding it are active.

WS objects allow us to encapsulate coordination necessary to reconfigure separately from the data objects stored in the system. These objects can be implemented in different ways, which allows for comparing different reconfiguration approaches in a modular fashion. Note, however, that previously proposed asynchronous WS algorithms [2, 3] are unsuitable for the data-centric model, since the storage overhead they require (when adapted naïvely to our model) is proportional to the number of clients, which can be arbitrarily large. We next present a new data-centric asynchronous WS algorithm, whose overhead is proportional only to the number of storage-nodes in the current system configuration, and is independent of the number of clients.

3.1 Asynchronous algorithm

Algorithm 1 is an asynchronous data-centric WS implementation. It uses a vector N_i of $|S|$ base-objects stored at each storage-node i, where $N_i[j]$ stores the value "endorsed" by storage-node j, if this value is known to storage-node i (initially $\perp$). The *update*(c) operation contacts a majority of the storage-nodes asking them to endorse the value c by invoking *compare&swap* on each storage-node in line 6. It then waits until the first storage-node, j, responds. Its response, *prev*, indicates whether j endorses c because it is the first such request it receives ($prev{=}\perp$), or whether $N_j[j]$ was not changed to c, and it is still equal to $prev{\neq}\perp$. The *update* then writes the value endorsed by j to $N_i[j]$ at a majority of storage-nodes (line 12). Note that all non-$\perp$ values that appear in $N_i[j]$ for the same j are identical, as *compare&swap* guarantees that each storage-node endorses at most one value. For the same reason, once a base-object gets a non-$\perp$ value, it is never modified.

The *scan* operation returns a set of values that were endorsed by storage-nodes. It invokes *collect*, which reads vectors N_i from a majority of the storage-nodes, and merges them into a single vector L in lines 23-24. The k-th entry in L is the value endorsed by storage-node k, if such a value appears in one or more of the vectors retrieved in line 21, or $\perp$ otherwise. Then, similarly to the *update* operation, *collect* has a write-back phase (lines 26-28) where it makes sure that a majority of the storage-nodes have the latest information, in this case, the non-$\perp$ values appearing in L (in line 27 we denote by $L\{I\}$ the sub-vector of L consisting of entries that correspond to indices in the set I). After invoking collect once, *scan* checks whether the returned set of values is empty. If so, it returns $\emptyset$, and otherwise invokes collect one more time. The second invocation is required to ensure that some common value appears in all non-empty sets returned by *scan* operations. In the full paper we prove that Algorithm 1 guarantees the properties of weak snapshots.

4. DYNADISK

Our data-centric reconfiguration system, DynaDisk, is based on the DynaStore algorithm [3]. The heart of DynaStore is the WS object, which encapsulates the coordination mechanism necessary for reconfigurations. In DynaDisk, each configuration w has a weak snapshot object WS(w) that stores reconfiguration proposals, i.e., changes proposed to w by clients. For each configuration, the associated storage-nodes keep several replicas of the stored data. Clients contact storage-nodes to read or write the data, or to learn about storage-nodes that were added or removed. If all the storage-nodes that a client knew about have been removed, then that client uses the discovery service to locate some suitable storage-nodes.

The protocol leverages the WS properties to guarantee that all clients observe read and write operations as if they happened in the same total order (which conforms with the operation precedence relation), despite crashes and concurrent reconfiguration requests. DynaDisk uses two alternative algorithms for implementing WS objects – an asynchronous algorithm presented in Section 3.1 and a second algorithm based on consensus.

In DynaDisk, as in DynaStore, each configuration's WS object may be updated by multiple clients. We can define a unique global sequence of configurations, as the sequence that starts with some fixed initial configuration, and continues by following the "first" proposal stored in each configuration's WS. Although this sequence is not visible to clients, they can read a superset of proposals, namely, the current set of values in the WS, which is guaranteed to contain the first proposed one. They then follow and read from *all* potential next configurations. Then, they write back the latest data they read to a final configuration, which is guaranteed to be part of the global sequence. In this way, even without consensus on the unique global sequence, the latest written data is guaranteed to be observed by a reader.

Besides the WS algorithm used by DynaStore, which is unsuitable for the data-centric model (see Section 3), most of DynaStore remaining logic can be easily adapted to the model. In particular, its read/write mechanism is based on the ABD algorithm [4]. The only other functionality of DynaStore that cannot work in our model is the broadcast of new configurations. This is required to inform all servers in a new configuration that it can now be used. In DynaDisk, this mechanism was replaced by an oracle, explained in Section 2, which encapsulates the necessary service-location functionality. When blocking on a *wait* statement longer than some threshold of time, a client starts querying the oracle periodically. If the oracle responds with a newer configuration the client restarts the operation in that configuration.

We stress that a location service is needed in *any* reconfigurable storage system. However, like other server-based (non-data-centric) solutions (e.g., [13]), DynaStore did not deal with the question of how clients can find the servers. In the data-centric model this question cannot be avoided, as the algorithm is now run by clients who must be able to find the storage.

Another important modification to DynaStore concerns adding support for multiple data objects. DynaDisk does not restrict the number of objects that an application can read and write, and at the same time, employs only one WS object per configuration. Supporting a large number of objects might pose a difficulty for clients that transfer the state during reconfigurations, as clients cannot be expected to have sufficient memory for copying all objects from the old configuration to the new one, e.g., if the entire storage-node's memory must be copied. To mitigate this, we added support for incremental state transfer, which copies objects one by one (or in small sets of limited size).

With incremental state transfer, it is possible that one object is copied to the new configuration c_1, but when a second object is transfered, a later configuration c_2 is found and the second object is written in c_2. When state-transfer is complete, *reconfig* "marks" the configuration to which the *first* object was transfered, c_1 in our example, as *ready*; all other objects can be found by following updates stored in WSs starting from the WS of configuration c_1. We implement *ready* as an atomic shared boolean object stored on the nodes in c_1 using the ABD algorithm [4]. Clients start their operations (read, write or *reconfig*) from some previously known ready configuration c, and whenever a new configuration c' is found, the client reads its *ready* object to determine whether it is safe to start subsequent operations in c' rather than c. The details of this mechanism, as well as other modifications and optimizations made to DynaStore, are deferred to the full paper.

5. IMPLEMENTATION AND EVALUATION

In order to compare the two approaches to reconfigurations we implemented DynaDisk's WS module in two ways: (1) using our asynchronous Algorithm 1; and (2) using the Active Disk Paxos consensus algorithm [7], with exponential backoff as the leader-election mechanism ($1ms.$ was used as minimal backoff). We used C# with the Microsoft CCR (Concurrency and Coordination Runtime) library. Clients use TCP to communicate with storage-nodes.

The storage overhead of consensus-based coordination is constant for each storage-node in each configuration (this stems from the properties of Active Disk Paxos), whereas the over-

head of Algorithm 1 is linear with the number of storage-nodes in the configuration.

We deployed the system on a cluster of 2-core 2GHz AMD Opteron machines connected by 1Gb Ethernet. We used 8 such machines – 6 for storage (2 storage-nodes per machine), and 2 for clients. In each experiment described below, all 12 storage-nodes are initially in the system. 5 clients concurrently perform sequences of *write* operations (with data size of 4KB). During each sequence, we start 1, 2, or 5 clients that simultaneously invoke *reconfig* to remove one of the storage-nodes.

We first examine the latency of *write* operations. Figure 1

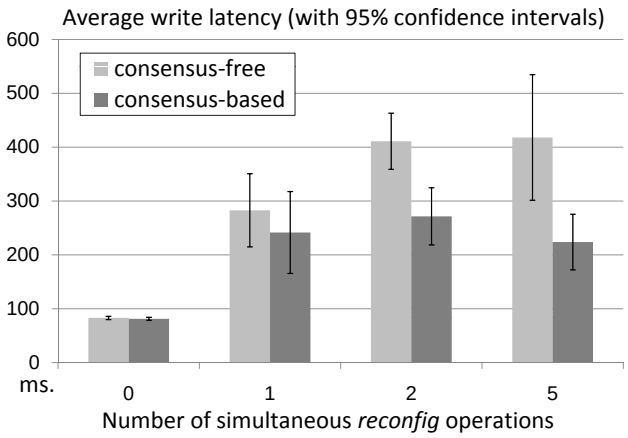

Figure 1: Average *write* latency.

shows that, as expected, in stable periods, when no *reconfig* is ongoing, the latency is the same whether we use consensus or not. When one or more reconfiguration are executing simultaneously with the *write*, we can see that the consensus-free approach has a noticeable negative effect on latency compared to the consensus-based approach. This happens because with consensus, even if multiple clients contend on who will become the leader and be the first to reconfigure the system, all other clients find out about changes to the configuration only after the consensus algorithm has reached a decision, and then, a single new configuration exists. On the other hand, in the consensus-free approach writers see all reconfiguration proposals as they are made and work with multiple possible configurations until a single configuration including all changes is formed.

We next examine the latency of *reconfig* operations in Figure 2. We see that when using consensus, average *reconfig* latency is slightly longer when many reconfigurations are in progress at the same time, and the variance of the time it takes to reconfigure is much bigger. This happens since multiple clients attempt to become leaders at the same time, which can lead to "infinite executions" in theory [9].

We have also emulated network latencies taken from distributions that model noisy LAN and WAN settings, as well as *read* operations instead of *write*s and experimented with adding storage-nodes instead of removing them. We got similar results, which are omitted for lack of space.

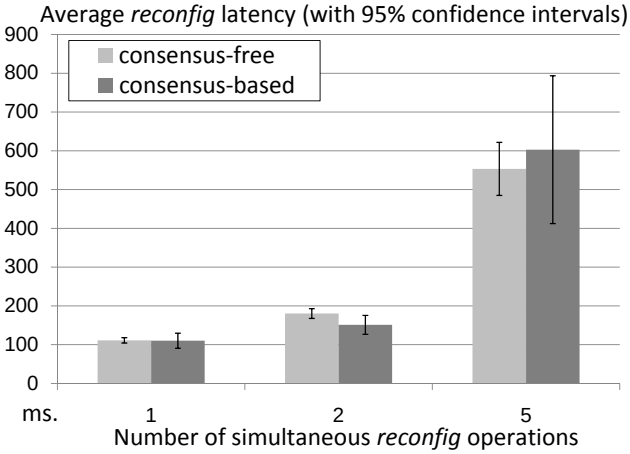

Figure 2: Average *reconfig* latency.

Acknowledgments

We thank Marcos K. Aguilera for many discussions and his insightful comments.

6. REFERENCES

[1] M. Abd-El-Malek, W. V. Courtright II, C. Cranor, G. R. Ganger, J. Hendricks, A. J. Klosterman, M. P. 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 *FAST*, pages 59–72, 2005.

[2] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. *Journal of the ACM*, 40(4):873–890, 1993.

[3] M. K. Aguilera, I. Keidar, D. Malkhi, and A. Shraer. Dynamic atomic storage without consensus. In *PODC*, pages 17–25, 2009.

[4] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1):124–142, 1995.

[5] K. Birman, G. Chockler, and R. van Renesse. Toward a cloud computing research agenda. *SIGACT News*, 40(2):68–80, 2009.

[6] G. Chockler, R. Guerraoui, I. Keidar, and M. Vukolic. Reliable distributed storage. *IEEE Computer*, 42(4):60–67, 2009.

[7] G. Chockler and D. Malkhi. Active disk paxos with infinitely many processes. *Distributed Computing*, 18(1):73–84, 2005.

[8] G. Chockler, D. Malkhi, and D. Dolev. A data-centric approach for scalable state machine replication. In FuDiCo, LNCS Volume 2584.

[9] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[10] S. Ghemawat, H. Gobioff, and S. T. Leung. The Google file system. In *SOSP*, pages 29–43, 2003.

[11] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A cost-effective, high-bandwidth storage architecture. *SIGOPS Operating Systems*

Review, 32(5):92–103, 1998.

[12] P. Jayanti, T. D. Chandra, and S. Toueg. Fault-tolerant wait-free shared objects. *Journal of the ACM*, 45(3), 1998.

[13] N. A. Lynch and A. A. Shvartsman. RAMBO: A reconfigurable atomic memory service for dynamic networks. In *DISC*, 2002.

[14] D. Malkhi and M. Reiter. Byzantine quorum systems. In *STOC*, pages 569–578, 1997.

[15] J.-P. Martin and L. Alvisi. A framework for dynamic Byzantine storage. In *DSN*, 2004.

[16] Y. Saito, S. Frølund, A. C. Veitch, A. Merchant, and S. Spence. Fab: Building distributed enterprise disk arrays from commodity components. In *ASPLOS*, 2004.