Fault Tolerance in Distributed Systems Using Fused Data Structures

Bharath Balasubramanian and Vijay K. Garg, *Fellow*, *IEEE*

Abstract—Replication is the prevalent solution to tolerate faults in large data structures hosted on distributed servers. To tolerate f crash faults (dead/unresponsive data structures) among n *distinct* data structures, replication requires $f+1$ replicas of each data structure, resulting in nf additional backups. We present a solution, referred to as *fusion* that uses a combination of erasure codes and selective replication to tolerate f crash faults using just f additional *fused backups*. We show that our solution achieves $O(n)$ savings in space over replication. Further, we present a solution to tolerate f Byzantine faults (malicious data structures), that requires only $nf+f$ backups as compared to the $2nf$ backups required by replication. We explore the theory of fused backups and provide a library of such backups for all the data structures in the Java Collection Framework. The theoretical and experimental evaluation confirms that the fused backups are space-efficient as compared to replication, while they cause very little overhead for normal operation. To illustrate the practical usefulness of fusion, we use fused backups for reliability in Amazon's highly available key-value store, Dynamo. While the current replication-based solution uses 300 backup structures, we present a solution that only requires 120 backup structures. This results in savings in space as well as other resources such as power.

Index Terms—Distributed systems, fault tolerance, data structures

1 Introduction

Distributed systems are often modeled as a set of independent servers interacting with clients through the use of messages. To efficiently store and manipulate data, these servers typically maintain large instances of data structures such as linked lists, queues, and hash tables. These servers are prone to faults in which the data structures may crash, leading to a total loss in state (crash faults [20]) or worse, they may behave in an adversarial manner, reflecting any arbitrary state, sending wrong conflicting messages to the client or other data structures (Byzantine faults [9]). *Active replication* [8], [21], [20] is the prevalent solution to this problem. To tolerate f crash faults among n given data structures, replication maintains $f+1$ replicas of each data structure, resulting in a total of nf backups. These replicas can also tolerate $\lfloor f/2 \rfloor$ Byzantine faults, since there is always a majority of correct copies available for each data structure. A common example is a set of lock servers that maintain and coordinate the use of locks. Such a server maintains a list of pending requests in the form of a queue. To tolerate three crash faults among, say five independent lock servers each hosting a queue, replication requires four replicas of each queue, resulting in a total of 15 backup queues. For large values of n, this is expensive in terms of the space required by the backups as well as power and other resources to maintain the backup processes. *Coding theory* [2], [14] is used as a space-efficient alternative to replication, both in the fields of communication and data storage. Data that needs to be transmitted across a channel is encoded using redundant bits that can correct errors introduced by a noisy channel [22], [5], [10]. Applications of coding theory in the storage domain include RAID disks [13] for persistent storage or information dispersal algorithms (IDAs) for fault tolerance in a set of data blocks [18]. In many large scale systems, such as Amazon's *Dynamo* key-value store [4], data is rarely maintained on disks due to their slow access times. The active data structures in such systems are usually maintained in main memory or RAM. In fact, a recent proposal of "RAMClouds" [12] suggests that online storage of data must be held in a distributed RAM, to enable fast access. In these cases, a direct application of coding-theoretic solutions, that are oblivious to the structure of data that they encode, is often wasteful. In the example of the lock servers, to tolerate faults among the queues, a simple coding-theoretic solution will encode the memory blocks occupied by the lock servers. Since the lock server is rarely maintained contiguously in main memory, a structure-oblivious solution will have to encode all memory blocks that are associated with the implementation of this lock server in main memory. This is not space efficient, since there could be a large number of such blocks in the form of free lists and memory book keeping information. Also, every small change to the memory map associated with this lock has to be communicated to the backup, rendering it expensive in terms of communication and computation.

In this paper, we present a technique referred to as *fusion* which combines the best of both these worlds to achieve the space efficiency of coding and the minimal update overhead of replication. Given a set of data structures, we maintain a set of *fused* backup data structures that can tolerate f crash faults among the given the data structures. In replication, the replicas for each data structure are identical to the given data structure. In fusion, the backup copies are not identical to the given data structures and hence, we make a distinction between the given data structures, referred to as *primaries* and

- *B. Balasubramanian is with the School of Engineering and Applied Sciences, Department of Electrical Engineering, Princeton University, Engineering Quadrangle, Olden Street, Princeton, NJ 08544. E-mail: bbharath@utexas.edu.*
- *V.K. Garg is with the Electrical and Computer Engineering Department, The University of Texas at Austin, Austin, TX 78731. E-mail: garg@ece.utexas.edu.*

Manuscript received 23 Aug. 2011; revised 8 Nov. 2011; accepted 3 Mar. 2012; published online 9 Mar. 2012.
Recommended for acceptance by J. Cao.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2011-08-0559.
Digital Object Identifier no. 10.1109/TPDS.2012.96.

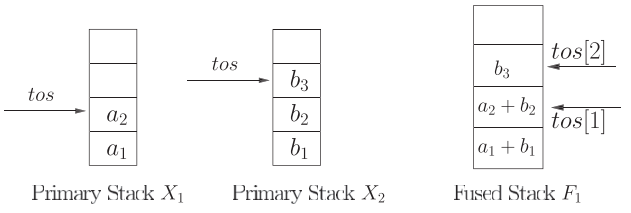


Fig. 1. Fault tolerant stacks.

the backup data structures, referred to as *backups*. Henceforth, in this paper, we assume that we are given a set of primary data structures among which we need to tolerate faults. Replication requires f additional copies of each primary ($f + 1$ replicas), resulting in nf backups. Fusion only requires f additional backups.

The fused backups maintain primary data in the coded form to save space, while they replicate the index structure of each primary to enable efficient updates. In Fig. 1, we show the fused backup corresponding to two primary array-based stacks X_1 and X_2. The backup is implemented as a stack whose nodes contain the sum of the values of the nodes in the primaries. We replicate the index structure of the primaries (just the top of stack pointers) at the fused stack. When an element a_3 is pushed on to X_1, this element is sent to the fused stack and the value of the second node (counting from zero) is updated to $a_3 + b_3$. In case of a pop to X_2, of say b_3, the second node is updated to a_3. These set of data structures can tolerate one crash fault. For example, if X_1 crashes, the values of its nodes can be computed by subtracting the values of the nodes in X_2 from the appropriate nodes of F_1. The savings in space is achieved by fusing the data nodes, while the index structure at the backups allows for efficient updates.

In Fig. 1, to tolerate one crash fault among X_1 and X_2, replication requires a copy for both X_1 and X_2, resulting in two backups containing five data nodes in total as compared to the fusion-based solution that requires just one backup containing three data nodes. When a crash fault occurs, recovery in replication just needs the state of the corresponding replica. Fusion on the other hand, needs all available data structures to decode the data nodes of the backups. This is the key tradeoff between replication and fusion. In systems with infrequent faults, the cost of recovery is an acceptable compromise for the savings in space achieved by fusion.

In [11], we present a coding-theoretic solution to fault tolerance in finite state machines. This approach is extended for infinite state machines and optimized for Byzantine fault tolerance in [6]. Our previous work on tolerating faults in data structures [7] provides the algorithms to generate a single fused backup for array or list-based primaries, that can tolerate one crash fault. In this paper, based on [23], we present a generic design of fused backups for most commonly used data structures such as stacks, vectors, binary search trees, hash maps and hash tables. Using erasure codes, we present f-fault tolerant data structures that tolerate f crash faults using just f additional fused backups. In the example, shown in Fig. 1, we can maintain another fused stack F_2 that has identical structure to F_1, but with nodes that contain the difference in values of the primary elements rather than the sum. These set of data structures can tolerate two crash faults. We extend this for values of f greater than two using Reed Solomon (RS) erasure codes [19], [3], [17], which are widely used to generate the optimal number of parity blocks in RAID-like systems.

Further, we consider the case of Byzantine faults, where the data structures can reflect arbitrary values, send conflicting erroneous responses to the client and try to maliciously defeat any protocol. Crash faults in a synchronous system, such as the one assumed in our model, can easily be detected using time outs. Detecting Byzantine faults is more challenging, since the state of the data structures need to be inspected on every update to ensure that there are no liars in the system. In this paper, we present a solution to tolerate f Byzantine faults among n primary data structures using just $nf + f$ backup structures as compared to the $2nf$ backups required by replication. We use a combination of replication and fusion to ensure minimal overhead during normal operation.

In addition, we prove properties on our fused backups such as space optimality, update optimality, and order independence. Given n primaries, our approach achieves $O(n)$ times savings in space over both replication and [7]. The time complexity of updates to our backups is identical to that for replication and $O(n)$ times faster than [7]. Similar to replication, we show that the updates to the backups can be done with a high level of concurrency. Further, we show that the updates to different backups from distinct primaries can be received in any order, thereby eliminating the need for synchronization at the backups.

In practical systems, sufficient servers may not be available to host all the backup structures and hence, some of the backups have to be distributed among the servers hosting the primaries. These servers can crash, resulting in the loss of all data structures residing on them. Consider a set of n data structures, each residing on a distinct server. We need to tolerate f crash faults among the servers given only a additional servers to host the backup structures. We present a solution to this problem that requires $\lceil n/(n + a - f)\rceil \cdot f$ backups and show that this is the necessary and sufficient number of backups for this problem. We also present a way to compare (or order) sets of backups of the same size, based on the number of primaries that they need to service. This is an important parameter because the load on a backup is directly proportional to the number of primaries it has to service. We show that our partitioning algorithm generates a minimal set of backups.

To illustrate the practical usefulness of fusion, we apply our design to Amazon's *Dynamo* [4], which is the highly available data-store underlying many of the services exposed by Amazon to the end-user. Examples include the service that maintains shopping cart information or the one that maintains user state. Dynamo achieves its twin goals of fault tolerance (durability) and fast response time for writes (availability) using a simple replication-based approach. We propose an alternate design using a combination of both fusion and replication, which requires far less space, while providing almost the same levels of durability, and availability for writes. In a typical host cluster, where there are 100 dynamo hosts each hosting a data structure, the current replication-based approach requires 300 backup

structures. Our approach, on the other hand, requires only 120 backup structures. This translates to significant savings in both the space occupied by the backups as well as the infrastructure costs such as power and resources required by the processes running these backups.

We provide a Java implementation of fused backups [1] using RS codes for all the data structures in the Java Collection Framework. Our experiments indicate that the current version of fusion is very space efficient as compared to both replication (approximately n times) and the older version (approximately $n/2$ times). The time taken to update the backups is almost as much as replication (approximately 1.5 times slower) while it is much better than the older version (approximately 2.5 times faster). Recovery is extremely cheap in replication but the current version of fusion performs approximately $n/2$ times better than the older version. Though recovery is costly in fusion as compared to replication, in absolute terms, it is still low enough to be practical (order of milliseconds). In the following section, we describe the system model of this paper.

2 Model and Notation

Our system consists of independent distributed servers hosting data structures. We denote the n given data structures, also referred to as primaries, $X_1 \dots X_n$. The backup data structures that are generated based on our idea of fusing primary data are referred to as *fused backups* or *fused data structures*. The operator used to combine primary data is called the *fusion operator*. The number of fused backups, t, depends on the fusion operator and the number of faults that need to be tolerated. The fused backups are denoted $F_1 \dots F_t$. In Fig. 1, X_1 and X_2 are the primaries, F_1 is the fused backup and the fusion operator is addition.

The data structures are modeled as a set of data nodes and an index structure that specifies order information about these data nodes. For example, the index structure for a linked list includes the head, tail, and next pointers. We assume that the size of data in the data structure far exceeds the size of its index structure. The data structures in our system have a *state* as well as an *output* associated with them. The state of a data structure is a snapshot of the values in the data nodes and the index structure. The output is the value visible to the external world or client. On application of an event/update the data structure transitions from one state to another and changes its output value. For example, the state associated with a linked list is the value of its nodes, next pointers, tail and head pointers. When we insert data into a linked list with a certain key, the value of the nodes and pointers change (state) and it responds with either success or failure (output).

We assume a single client that sends updates to the various data structures and acts on the responses/output received from them. When an update is sent to a primary data structure, the primary first updates itself and then sends sufficient information to update the backups. We assume FIFO communication channels that are reliable and have a strict upper bound on time for all message delivery, i.e., a synchronous system. Hence, all updates to the data structures from the client, is received in the same order at the data structures. The problem of multiple clients sending updates to the data structures, which may be received in any order, is beyond the scope of this paper.

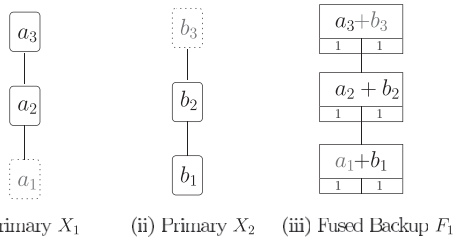


Fig. 2. Old fusion [7].

Faults among the data structures, both primaries and backups, can be of two types: *crash* faults (also referred to as *fail-stop* faults) and *Byzantine* faults. In the case of crash faults, the data structure crashes and stops responding to the client, leading to a complete loss in state. For Byzantine faults, the data structure can assume arbitrary values for its state, send wrong responses to the client/other data structures and in general behave maliciously to defeat any protocol. However, the data structure cannot fake its identity. Since active replication is a fault-masking technique, so far in this paper, we have said that replication can tolerate faults among the data structures. However, for fusion, we need to decode the values and correct the faults in the system. Henceforth, for convenience, we say that backups (for both replication and fusion) "correct" faults among primaries.

Detection and correction of faults in our system is performed by the fault-free client. Since we assume a synchronous system, crash faults are detected using timeouts. If a data structure does not respond to an update sent by the client within a fixed time period, it is assumed to have crashed. We present algorithms for the detection of Byzantine faults. When a fault occurs, no updates are sent by the client until the state of all the failed data structures have been recovered. For recovery, the client acquires the state of the requisite data structures after they have acted on all updates before the fault occurred, and then recovers the state of the failed structures. Henceforth, when we simply say faults, we refer to crash faults. The design of the fused data structures is independent of the fault model and for simplicity we explain the design assuming only crash faults.

3 Fusion-Based Fault Tolerant Data Structures

Design motivation. In [7], Garg and Ogale present a design to fuse array and list-based primaries that can correct one crash fault. We highlight the main drawback of their approach for linked lists. The fused structure for linked list primaries in [7] is a linked list whose nodes contain the XOR (or sum) of the primary values. Each node contains a bit array of size n with each bit indicating the presence of a primary element in that node. A primary element inserted in the correct position at the backup by iterating through the fused nodes using the bit array and a similar operation is performed for deletes. An example is shown in Fig. 2 with two primaries and one backup. After the delete of primary elements a_1 and b_3, the first and third nodes of the fused

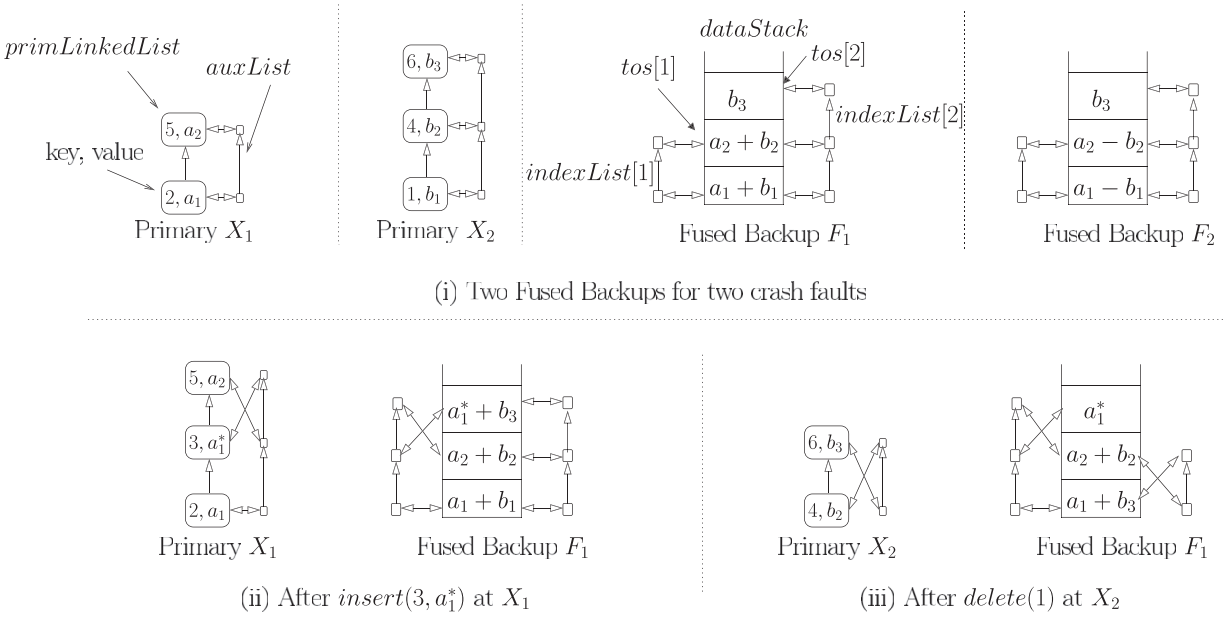


Fig. 3. Fused backups for linked lists (keys not shown in F_1, F_2 due to space constraint).

backup F_1 are updated to b_1 and a_3, respectively (deleted elements in grayscale). After the deletes, while the primaries each contain only two nodes, the fused backup contains three nodes. If there are a series of inserts to the head of X_1 and to the tail of X_2 following this, the number of nodes in the fused backup will be very high. This brings us to the main design motivation of this section: Can we provide a generic design of fused backups, for all types of data structures such that the fused backup contains only as many nodes as the largest primary, while guaranteeing efficient updates? We present a solution for linked lists and then generalize it for complex data structures.

3.1 Fused Backups for Linked Lists

We use a combination of replication and erasure codes to implement fused backups each of which are identical in structure and differ only in the values of the data nodes. In our design of the fused backup, we maintain a stack of nodes, referred to as *fused nodes* that contains the data elements of the primaries in the coded form. The fused nodes at the same position across the backups contain the same primary elements and correspond to the code words of those elements. Fig. 3 shows two primary sorted linked lists X_1 and X_2 and two fused backups F_1 and F_2 that can correct two faults among the primaries. The fused node in the 0th position at the backups contain the elements a_1 and b_1 with F_1 holding their sum and F_2 their difference. At each fused backup, we also maintain index structures that replicate the ordering information of the primaries. The index structure corresponding to primary X_i is identical in structure to X_i, but while X_i consists of data nodes, the index structure only contains pointers to the fused nodes. The savings in space are achieved because primary nodes are being fused, while updates are efficient since we maintain the index structure of each primary at the backup.

Overview. We begin with a high-level description on how we restrict the number of nodes in the backup stack. At each backup, elements of primary X_i are simply inserted one on top of the other in the stack with a corresponding update to the index structure to preserve the actual ordering information. The case of deletes is more complex. If we just delete the element at the backup, then similar to Fig. 2, a "hole" is created and the fused backups can grow very large. In our solution, we shift the top-most element of X_i in the backup stack, to plug this hole. This ensures that the stack never contains more nodes than the largest primary. Since the top-most element is present in the fused form, the primary has to send this value with every delete to enable this shift. To know which element to send with every delete, the primary has to track the order of its elements at the backup stack. We achieve this by maintaining an auxiliary list at the primary, which mimics the operations of the backup stack. When an element is inserted into the primary, we insert a pointer to this element at the end of its auxiliary list. When an element is deleted from the primary, we delete the element in the auxiliary list that contains a pointer to this element and shift the final auxiliary element to this position. Hence, the primary knows exactly which element to send with every delete. Fig. 3 illustrates these operations with an example. We explain them in greater detail in the following paragraphs.

Inserts. Fig. 4 shows the algorithms for the insert of a key-value pair at the primaries and the backups. When the client sends an insert to a primary X_i, if the key is not already present, X_i creates a new node containing this key-value, inserts it into the primary linked list (denoted $primaryLinkedList$) and inserts a pointer to this node at the end of the aux list ($auxList$). The primary sends the key, the new value to be added and the old value associated with the key to all the fused backups. Each fused backup maintains a stack ($dataStack$) that contains the primary elements in the coded form. On receiving the insert from X_i, if the key is not already present, the backup updates the code value of the fused node following the one contains the top-most element of X_i (pointed to by $tos[i]$). To maintain

Insert at Primaries $X_i :: i = 1..n$
Input: key k, data value d;
Var: Linked List $primaryLinkedList$ (given data structure), $auxList$ (order of data at the backup);
if $(primaryLinkedList \cdot contains(k))$
/* key present, just update its value*/
$old = primaryLinkedList \cdot get(k) \cdot value$
$primaryLinkedList \cdot update(k, d)$;
$send(k, d, old)$ to all fused backups;
else
/* key not present, create new node*/
primNode p = new primNode;
$p \cdot value = d$;
auxNode a = new auxNode;
$a \cdot primNode = p$;
$p \cdot auxNode = a$;
/* mimic backup stack */
$auxList.insertAtEnd(a)$;
$primaryLinkedList \cdot insert(k, p)$;
$send(k, d, null)$ to all fused backups;

Insert at Fused Backups $F_j :: j = 1..t$
Input: key k, new value d_i, old value old_i;
Var: Stack $dataStack$ (stack of fused nodes), Linked Lists[] $indexList[n]$ (order of primary data), Pointer to Fused Nodes[] $tos[n]$ (top of stack pointers);
if $(indexList[i] \cdot contains(k))$
fusedNode $f = indexList[i] \cdot get(k)$;
$f \cdot updateCode(old_i, d_i)$;
else
fusedNode $p = tos[i] + +$;
if $(p == null)$
p = new fusedNode;
$dataStack \cdot push(p)$;
$p \cdot updateCode(0, d_i)$;
$p \cdot refCount + +$;
/* mimic primary linked list */
indexNode $a =$ new indexNode;
$a \cdot fusedNode = p$;
$p \cdot indexNode[i] = a$;
$indexList[i] \cdot insert(k, a)$;

Fig. 4. Fused backups for linked lists: inserts.

Delete at Primaries $X_i :: i = 1..n$
Input: key k;
Var: Linked List $primaryLinkedList$ (given data structure), $auxList$ (order of data at the backup);
$p = primaryLinkedList \cdot delete(k)$;
$old = p \cdot value$;
/* tail node of aux list points to top-most element of X_i at backup stack */
auxNode $auxTail = auxList \cdot getTail()$;
$tos = auxTail \cdot primNode \cdot value$;
$send(k, old, tos)$ to all fused backups;
auxNode $a = p \cdot auxNode$;
/* shift tail of aux list to replace a */
$(a \cdot prev) \cdot next = auxTail$;
$auxTail \cdot next = a \cdot next$;
$delete\ a$;

Delete at Fused Backups $F_j :: j = 1..t$
Input: key k, old value old_i, end value tos_i;
Var: Stack $dataStack$ (stack of fused nodes), Linked Lists[] $indexList[n]$ (order of primary data), Pointer to Fused Nodes[] $tos[n]$ (top of stack pointers);
/* update fused node containing old_i with primary element of X_i at $tos[i]$*/
indexNode $a = indexList[i] \cdot delete(k)$;
fusedNode $p = a \cdot fusedNode$;
$p \cdot updateCode(old_i, tos_i)$;
$tos[i] \cdot updateCode(tos_i, 0)$;
$tos[i] \cdot refCount - -$;
/* update index node pointing to $tos[i]$ */
$tos[i] \cdot indexNode[i] \cdot fusedNode = p$;
if $(tos[i].refCount == 0)$
$dataStack.pop()$;
$tos[i] - -$;

Fig. 5. Fused backups for linked lists: deletes.

order information, the backup inserts a pointer to the newly updated fused node, into the index structure ($indexList[i]$) for X_i with the key received. A reference count ($refCount$) tracking the number of elements in the fused node is maintained to enable efficient deletes.

Fig. 3ii shows the state of X_1 and F_1 after the insert of $(3, a_1^*)$. We assume that the keys are sorted in this linked list and hence the key-value pair $(3, a_1^*)$ is inserted at index 1 of the primary linked list and a pointer to a_1^* is inserted at the end of the aux list. At F_1, the value of the second node (nodes numbered from zero) is updated to $a_1^* + b_3$ and a pointer to this node is inserted at index 1 of $indexList[1]$. The identical operation is performed at F_2 (not shown in the figure due to space constraints), with the only difference being that the second fused node is updated to $a_1^* - b_3$. Observe that the aux list at X_1 specifies the exact order of elements maintained at the backup stack ($a_1 \rightarrow a_2 \rightarrow a_1^*$). Analogously, $indexList[1]$ at the fused backup points to the fused nodes that contain elements of X_1 in the correct order ($a_1 \rightarrow a_1^* \rightarrow a_2$).

Delete. Fig. 5 shows the algorithms for the delete of a key at the primaries and the backups. X_i deletes the node associated with the key from the primary and obtains its value which needs to be sent to the backups. Along with this value and the key k, the primary also sends the value of the element pointed by the tail node of the aux list. This corresponds to the top-most element of X_i at the backup stack and is hence required for the shift operation that will be performed at the backup. After sending these values, the primary shifts the final node of the aux list to the position of the aux node pointing to the deleted element, to mimic the shift of the final element at the backup.

At the backup, since $indexList[i]$ preserves the exact order information of X_i, by a simple double dereference, we can obtain the fused node p that contains the element of X_i associated with k. The value of p is updated with the top-most element (sent by the primary as tos) to simulate the shift. The pointers of $indexList[i]$ are updated to reflect this shift. Fig. 3iii shows the state of X_1 and F_1 after the delete of b_1. The key facts to note are: 1) at F_1, b_3 has been shifted from the end to the 0th node, 2) the aux list at X_2 reflects the correct order of its elements at the backup stack ($b_3 \rightarrow b_2$), and 3) $indexList[2]$ reflects the correct order of elements at X_2 ($b_2 \rightarrow b_3$). In Appendix A, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2012.96,

we extend this design to complex data structures such as maps, trees, and hash tables.

As specified in Section 2, we assume that the size of the data far exceeds the size of the auxiliary structure. For example, in Fig. 3, if the size of the primary or fused nodes is in the order of megabytes, the size of the index structures or auxiliary structures is just in the order of bytes (next pointers). So the space overhead of maintaining these auxiliary/index structures is negligible. Also, the auxiliary structures at the primary can be updated in constant time for both inserts and deletes with the use of double-ended pointers. Hence, they do not cause any additional overhead in terms of time.

So far, we have used simple sum-difference as the fusion operator, that can correct two crash faults using two backups. Given two integers, say a and b, it is sufficient to maintain the integers $(a+b)$ and $(a-b)$ to correct two faults among these four integers. Similarly, it can be seen that, to correct three faults among three integers a, b, and c, it is sufficient to maintain the integers $(a+b+c)$, $(a+2b+3c)$, and $(a+4b+9c)$. In Appendix B, which is available in the online supplemental material, we generalize this and present the Reed Solomon erasure codes that can be used as a fusion operator to correct f crash faults among the primaries using f backups. While the structure of all the f fused backups will be identical, the fused nodes will contain the RS checksum values of the primary elements. The key intuition behind RS codes is to form f checksums from n given data elements, such that despite f failures among the data and checksums, we have sufficient linearly independent equations to recover the failed data.

3.2 Correcting Crash Faults

To correct crash faults, the client needs to acquire the state of all the available data structures, both primaries and backups. As seen in Section 3.1, the fused node at the same position at all the fused backups are the codewords for the primary elements belonging to these nodes. To obtain the missing primary elements belonging to this node, we decode the code words of these nodes along with the data values of the available primary elements belonging to this node. The decoding algorithm depends on the erasure code used. In Fig. 3(i), to recover the state of the failed primaries, we obtain the states F_1 and F_2 and iterate through their nodes. The 0th fused node of F_1 contains the value a_1+b_1, while the 0th node of F_2 contains the value a_1-b_1. Using these, we can obtain the values of a_1 and b_1. The value of all the primary nodes can be obtained this way and their order can be obtained using the index structure at each backup. In Appendix A, which is available in the online supplemental material, we show that the time complexity of recovery using RS codes as the fusion operator is $O(nmst^2)$, given n primaries with $O(m)$ nodes of $O(s)$ size each, with t actual crash faults among them ($t \leq f$). Recovery is much cheaper in replication and has time complexity $O(mst)$.

4 Theory of Fused Data Structures

In this section, we prove properties on the fused backups such as size optimality, update optimality, and update order independence, all of which are important considerations when implementing a system using these backups. These properties ensure that the overhead in space and time caused due to these backups is minimal. The results in this section apply for all types of primaries and are independent of the fusion operator used. The only assumption we make is that the codes can be updated locally in constant time (like updates in RS codes).

4.1 Space Optimality

Consider n primaries, each containing $O(m)$ nodes, each of size $O(s)$. In [7], to correct one crash fault, the backup for linked lists and list-based queues consumes $O(nms)$ space, which is as bad as replication. We show that the fused backups presented in this paper require only $O(ms)$ space. Further, to correct f faults, we show that the fused backups need only $O(msf)$ space. Replication, on the other hand requires $O(mnsf)$ space, which is $O(n)$ times more than fusion. To correct f crash faults, we use RS codes that require f fused backups, which is the minimum number of backups required for f faults. For example, in Fig. 3, the number of fused nodes in F_1 or F_2 is always equal to the number of nodes in the largest primary. The optimal size of the data stack in our backups combined with RS codes as the fusion operator, leads to the result that our solution is space optimal when the data across the primaries is uncorrelated.

Theorem 1 (Space Optimality). *The fused backups generated by our design using RS codes as the fusion operator are of optimal size.*

Proof. We first show that the data stack of each backup contains only m fused nodes. A *hole* is defined as a fused node that does not contain an element from a primary followed by a fused node that contains an element from that primary. When there are no holes in the data stack, each primary element is stacked one on top of the other and the stack contains only m nodes, i.e., as many nodes as the largest primary. We maintain the invariant that our data stack never has holes.

In inserts to X_i, we always update the fused node on top of the last fused node containing an element from X_i. Hence, no hole is created. For deletes, when a hole is created, we shift the final element of the primary, pointed by $tos[i]$ to plug this hole If the size of each node is $O(s)$, then the backup space required by our solution to correct f crash faults is $O(msf)$.

Now, f crash faults among the primaries will result in the failure of at least f data nodes, each of size $O(s)$. To correct f crash faults among them, any solution needs to maintain at least f backup nodes each of size $O(s)$. Since the data structures each contain $O(m)$ nodes, to correct f crash faults among them, any solution needs to maintain f backups containing each containing $O(ms)$ space. Hence, the minimum space required is $O(msf)$. □

4.2 Efficient Updates

We define update optimality as follows: the time complexity of updates at any fused backup for all operations is the same as that of the corresponding update to the primary. In [7], to update the backup for linked lists, we need to iterate through all the fused nodes. Since the number of fused nodes in the backup is $O(nm)$, the time complexity of updates is $O(nm)$. However, since each primary linked list

has $O(m)$ nodes, update to a primary takes only $O(m)$ time. Hence, this solution is not update optimal. We show that the fused backups presented in this paper are update optimal for all types of primaries. So, fusion causes has same minimal overhead during normal operation as replication. Our proof is based on the following simple intuition. The time complexity of update to the primaries depends on its index structure. For example, in the case of a linked list the index structure consists of next pointers. So to update a linked list with $O(m)$ nodes it takes $O(m)$ time. Since we replicate the index structure of each primary completely at the backup, the time complexity of the update to the fused backup is same as that at the primary.

Theorem 2 (Update Optimality). *The time complexity of the updates to a fused backup is of the same order as that at the primary.*

Proof. In the case of inserts, we obtain the node following the top most element of X_i in the data stack and update it in constant time. The update to the index structure consists of an insert of an element with key k, which is the identical operation at the primary. Similarly, for deletes, we first remove the node with key k from the index structure, an operation that was executed on the data structure of the same type at the primary. Hence, it takes as much time as that at the primary. Shifting the final element of this primary to the fused node that contains the deleted element is done in constant time.

This argument for inserts and deletes extends to more complex operations: any operation performed on the primary will also be performed on the index structure at the backup. Updating the data nodes of the stack takes constant time. □

Since the primaries are independent of each other, in many cases the updates to the backup can be to different fused nodes. In the following theorem, we show that multiple threads belonging to different primaries can update the fused backups by locking just a constant number of nodes. Hence, fusion can achieve considerable speed-up.

Theorem 3 (Concurrent Updates). *There exists an algorithm for multiple threads belonging to different primaries to update a fused backup concurrently by locking just a constant number of nodes.*

Proof. We modify the algorithms in Figs. 4 and 5 to enable concurrent updates. We assume the presence of fine grained locks that can lock just the fused nodes and if required a fused node along with the $dataStackTos$. Since updates from the same primary are never applied concurrently, we don't need to lock the index structure.

Inserts. If the insert to the fused backup has to create a new fused node, then the updating thread has to lock $dataStackTos$ and the fused node pointed by this pointer using a single lock, insert and update a new fused node, increment $dataStackTos$ and then release this combined lock. If the insert from X_i does not have to create a new node it only has to lock the fused node pointed by $tos[i]$, update the node's code value and release the lock. When the primaries are of different sizes, then the insert to the backups never occurs to the same fused node and hence are fully concurrent.

Deletes. The updating thread has to obtain the fused node containing the element to be deleted, lock it, update its value and release it. Then it has to lock the node pointed by $tos[i]$, update its value and release the lock. Similar to the case of inserts, when the delete causes a node of the stack to be deleted, the thread needs to lock the $dataStackTos$ as well as the node pointed by this pointer in one lock, delete the node, update the pointer, and then release the combined lock. □

4.3 Order Independence

In the absence of any synchronization at the backups, updates from different primaries can be received in any order at the backups. The assumption of FIFO communication channels only guarantees that the updates from the *same primary* will be received by all the backups in the same order. A direct extension of the solution in [7] for multiple faults can result in a state from which recovery is impossible. For example, in Fig. 3, F_1 may receive the insert to X_1 followed by the delete to X_2 while F_2 may receive the delete update followed by the insert. To achieve recovery, it is important that the fused nodes at the same position at different fused backups contain the same primary elements (in different coded forms). In Fig. 3(i), if the 0th node of F_1 contains $a_1 + b_1$, while the 0th node of F_2 contains $a_2 - b_1$, then we cannot recover the primary elements when X_1 and X_2 fail.

We show that in the current design of fused backups, the nodes in the same position across the fused backups always contain the same primary elements independent of the order in which the updates are received at the backups. Also, the index structures at the backups are also independent of the order in which the updates are received. Consider the updates shown in Fig. 3. The updates to the index lists commute since they are to different lists. As far as updates to the stack are concerned, the update from X_1 depends only on the last fused node containing an element from X_1 and is independent of the update from X_2 which does not change the order of elements of X_1 at the fused backup. Similarly the update from X_2 is to the first and third nodes of the stack immaterial of whether a_1^* has been inserted.

Theorem 4 (Order Independence). *The state of the fused backups after a set of updates is independent of the order in which the updates are received, as long as updates from the same primary are received in FIFO order.*

Proof. Clearly, updates to the index structure commute. As far as updates to the stack are concerned, the proof follows from two facts about our design. First, updates on the backup for a certain primary do not affect the order of elements of the other primaries at the backup. Second, the state of the backup after an update from a primary depends only on the order of elements of that primary. The same argument extends to other complex operations that only affect the index structure. □

4.4 Fault Tolerance with Limited Backup Servers

So far we have implicitly assumed that the primary and backup structures reside on independent servers for the

fusion-based solution. In many practical scenarios, the number of servers available maybe less than the number of fused backups. In these cases, some of the backups have to be distributed among the servers hosting the primaries. Consider a set of n data structures, each residing on a distinct server. We need to correct f crash faults among the servers given only a additional servers to host the backup structures. We present a solution to this problem that requires $\lceil n/(n+a-f)\rceil \cdot f$ backups and show that this is the necessary and sufficient number of backups for this problem. Further, we present an algorithm for generating the optimal number of backups.

To simplify our discussion, we start with the assumption that *no* additional servers are available for hosting the backups. As some of the servers host more than one backup structure, f faults among the servers, results in more than f faults among the data structures. Hence, a direct fusion-based solution cannot be applied to this problem. Given a set of five primaries, $\{X_1 \dots X_5\}$, each residing on a distinct server labelled, $\{H_1 \dots H_5\}$, consider the problem of correcting three crash faults among the servers ($n = 5$, $f = 3$). In a direct fusion-based solution, we will just generate three backups F_1, F_2, and F_3, and distribute them among any three servers, say, H_1, H_2, and H_3, respectively. Crash faults among these three servers will result in the crash of six data structures, whereas these set of backups can only correct three crash faults. We solve this problem by partitioning the set of primaries and generating backups for each individual block.

In this example, we can partition the primaries into three blocks $[X_1, X_2]$, $[X_3, X_4]$, and $[X_5]$ and generate three fused backups for each block of primaries. Henceforth, we denote the backup obtained by fusing the primaries $X_{i_1}, X_{i_2}, \dots$, by $F_j(i_1, i_2, \dots)$. For example, the backups for $[X_1, X_2]$ are denoted as $F_1(1,2) \cdots F_3(1,2)$. Consider the following distribution of backups among hosts:

$$H_1 = [X_1, F_1(3,4), F_1(5)], H_2 = [X_2, F_2(3,4), F_2(5)],$$
$$H_3 = [X_3, F_1(1,2), F_3(5)], H_4 = [X_4, F_2(1,2)],$$
$$H_5 = [X_5, F_3(1,2), F_3(3,4)].$$

Note that, the backups for any block of primaries, do not reside on any of the servers hosting the primaries in that block. Three server faults will result in at most three faults among the primaries belonging to any single block and its backups. Since the fused backups of any block correct three faults among the data structures in a block, this partitioning scheme can correct three server faults.

For example, assume crash faults in the servers H_2, H_4, and H_5. Consider the recovery of X_2 on the crashed server, H_2. Since, $F_1(1,2), F_2(1,2), F_3(1,2)$ are the three fused backups for $[X_1, X_2]$, given the state of any two data structures among $\{X_1, X_2, F_1(1,2), F_2(1,2), F_3(1,2)\}$, we can recover the state of the remaining three. In our example, we can obtain the state of X_1 on server H_1, and the state of $F_1(1,2)$ on server H_3 (servers that have not crashed). Given the state of these two data structures we can recover the state of X_2, $F_2(1,2)$, and $F_3(1,2)$. Here, each block of primaries requires at least three distinct servers (other than those hosting them) to host their backups. Hence, for $n = 5$, the size of any block in this partition cannot exceed $n - f = 2$. Based on this idea, we present an algorithm to correct f faults among the servers.

Partitioning algorithm. Partition the set of primaries X as evenly possible into $\lceil n/(n-f)\rceil$ blocks, generate the f fused backups for each such block and place them on distinct servers not hosting the primaries in that block.

The number of blocks generated by the partitioning algorithm is $\lceil n/(n-f)\rceil$ and hence, the number of backup structures required is $\lceil n/(n-f)\rceil \cdot f$. Replication, on the other hand requires $n \cdot f$ backup structures which is always greater than or equal to $\lceil n/(n-f)\rceil \cdot f$. We show that $\lceil n/(n-f)\rceil \cdot f$ is a tight bound for the number of backup structures required to correct f faults among the servers. For the example where $n = 5$, $f = 3$, the partitioning algorithm requires nine backups. Consider a solution with eight backups. In any distribution of the backups among the servers, the three servers with the maximum number of data structures will host nine data structures in total. For example, if the backups are distributed as evenly as possible, the three servers hosting the maximum number of backups will each host two backups and a primary. Failure of these servers will result in the failure of nine data structures. Using just eight backups, we cannot correct nine faults among the data structures. We generalize this result in the following theorem.

Theorem 5. *Given a set of n data structures, each residing on a distinct server, to correct f crash faults among the servers, it is necessary and sufficient to add $\lceil n/(n+a-f)\rceil \cdot f$ backup structures, when there are only a additional servers available to host the backup structures.*

Proof. We first prove sufficiency, followed by the proof showing that it is necessary to maintain that many backups.

Sufficiency. We modify the partitioning algorithm for a additional servers simply by partitioning the primaries into $\lceil n/(n+a-f)\rceil$ blocks rather than $\lceil n/(n-f)\rceil$ blocks. Since the maximum number of primaries in any block of the partitioning algorithm is $n+a-f$, there are at least f distinct servers (not hosting the primaries in the block) available to host the f fused backups of any block of primaries. So, the fused backups can be distributed among the host servers such that f server faults only lead to f faults among the backups and primaries corresponding to each block. Hence, the fused backups generated by the partitioning algorithm can correct f server faults.

Necessity. Suppose there is a scheme with t backups such that $t < \lceil n/(n+a-f)\rceil \cdot f$. In any distribution of the backups among the servers, choose f servers with the largest number of backups. We claim that the total number of backups in these f servers is strictly greater than $t - f$. Failure of these servers, will result in more than $t - f + f$ faults (adding faults of f primary structures). This would be impossible to correct with t backups. We know that,

$$\begin{aligned} t &< \lceil n/(n+a-f)\rceil \cdot f \\ &\Rightarrow t < \lceil 1 + f/(n+a-f)\rceil \cdot f \\ &\Rightarrow (t-f) < \lceil f/(n+a-f)\rceil \cdot f \\ &\Rightarrow (t-f)/f < \lceil f/(n+a-f)\rceil. \end{aligned}$$

If the f servers with the largest number of backups have less than or equal to $t - f$ backups in all, then the

server with the smallest number of backups among them will have less than the average number of backups which is $(t-f)/f$. Since the remaining $n+a-f$ servers have more than or equal to f backups, the server with the largest number of backups among them will have as many or greater than the average number of backups, $\lceil f/(n+a-f)\rceil$. Since, $(t-f)/f < \lceil f/(n+a-f)\rceil$, we get a contradiction that the smallest among the f servers hosting the largest number of backups, hosts less number of backups than the largest among the remaining $n-f$ servers. □

4.4.1 Minimality

In this section, we define a partial order among equal sized sets of backups and prove that the partitioning algorithm generates a *minimal* set of backups.

Given a set of four data structures, $\{X_1 \dots X_4\}$, each residing on a distinct server, consider the problem of correcting two faults among the servers, with no additional backup servers ($n=4, f=2, a=0$). Since, $\lceil n/(n+a-f)\rceil = 2$, the partitioning algorithm will partition the set of primaries into two blocks, say $[X_1, X_2]$ and $[X_3, X_4]$ and generate four fused backups, $F_1(1,2)$, $F_2(1,2)$ and $F_1(3,4)$, $F_2(3,4)$. An alternate solution to the problem is to fuse the entire set of primaries to generate four fused backups, $F_1(1,2,3,4) \dots F_4(1,2,3,4)$. Here, $F_1(1,2)$ is obtained by fusing the primaries X_1 and X_2, whereas $F_1(1,2,3,4)$ is obtained by fusing all four primaries. In the latter case, maintenance is more expensive, since the backups need to receive and act on updates corresponding to all the primaries, whereas in the former, each backup receives inputs corresponding to just two primaries. Based on this idea, we define an order among backups. Given a set of n data structures, X, consider backups F and F', obtained by fusing together a set of primaries, $M \subseteq X$ and $N \subseteq X$, respectively. F is less than F' ($F < F'$) if $M \subseteq N$. In the example discussed, $F_1(1,2) < F_1(1,2,3,4)$, as $\{X_1, X_2\} \not\subseteq \{X_1, X_2, X_3, X_4\}$. We extend this to define an order among sets of backups that correct f faults among the servers.

Definition 1 (Order among Sets of Backups). *Given a set of n data structures, each residing on a distinct server, consider two sets of t backups, Y and Y′ that correct f faults among the servers. Y is less than Y′, denoted $Y < Y'$, if the backups in Y can be ordered as $\{F_1, \dots, F_t\}$ and the backups is Y′ can be ordered as $\{F'_1, \dots, F'_t\}$ such that $(\forall 1 \le i \le t : F_i \le F'_i) \wedge (\exists j : F_j < F'_j)$.*

A set of backups Y is *minimal* if there exists no set of backups Y' such that $Y' < Y$.

In the example for $n=4$, $f=2$, the set of backups, $Y = \{F_1(1,2), F_2(1,2), F_1(3,4), F_2(3,4)\}$, generated by the partitioning algorithm is clearly less than the set of backups, $Y' = \{F_1(1,2,3,4) \cdots F_4(1,2,3,4)\}$. We show that the partitioning algorithm generates a minimal set of backups.

Theorem 6. *Given a set of n data structures, each residing on a distinct server, to correct f faults among the servers, the partitioning algorithm generates a minimal set of backups.*

Proof. When a backup F is generated by fusing together a set of primaries, we say that each primary in the set *appears* in the backup. Given a set of backups that can tolerate f faults among the servers, each primary has to appear at least f times across all the backups. The partitioning algorithm generates a set of backups Y_p, in which each primary appears exactly f times. Any other solution in which the primaries appear exactly f times will be incomparable to Y_p. □

5 Detection and Correction of Byzantine Faults

So far, in this paper, we have only assumed crash faults. We now discuss Byzantine faults where any data structure may change its state arbitrarily, send wrong conflicting messages to the client/other data structures and in general attempt to foil any protocol. However, we assume that the data structures cannot fake their identity. To correct f Byzantine faults among n primaries pure replication requires $2f$ additional copies of each primary, which ensures that a nonfaulty majority of $f+1$ copies are always available. Hence, the correct state of the data structure can easily be ascertained. This approach requires $2nf$ backup data structures in total. Recovery in replication reduces to finding the state with $t+1$ votes among the $2f+1$ copies of each primary, where t is the actual number of faults. Since this majority can be found by inspecting at most $2t+1$ copies among the primaries, recovery has time complexity $O(mst)$, where m is the number of nodes in each data structure and s is the size of each data structure.

In this section, we present a hybrid solution that combines fusion with replication to correct f Byzantine faults using just $nf+f$ backup structures, while ensuring minimal overhead during normal operation. Recovery is costlier in fusion, with time complexity $O(msft^2 + nst^2)$. The algorithms and proofs in this section are an extension of the results in [6], which focuses on fault tolerance in infinite state machines.

In our solution, we maintain f additional copies of each primary that enable efficient *detection* of Byzantine faults. This maintains the invariant that there is at least one correct copy in spite of f Byzantine faults. We also maintain f fused backups for the entire set of primaries, which is used to identify and *correct* the Byzantine primaries, after the detection of the faults. Thus, we have a total of $nf+f$ backup data structures. The only requirement on the fused backups $\{F_j, j = 1 \cdots f\}$ is that if F_j is not faulty, then given the state of any $n-1$ data structures among $\{X_1 \cdots X_n\}$, we can recover the state of the missing one. Thus, a simple XOR or sum based fused backup is sufficient. Even though we are correcting f faults, the requirement on the fused copy is only for a single fault (because we are also using replication).

The primary X_i and its f copies are called *unfused* copies of X_i. If any of the $f+1$ unfused copies differ, we call the primary, *mismatched*. Let the state of one of the unfused copies (which includes the value of the data elements, auxiliary structure and index information) be v. The number of unfused copies of X_i with state v is called the *multiplicity* of that copy.

Unfused Copies:
On receiving any message from client
Update local copy;
send state update to fused processes;
send response to the client;

Client:
send update to all unfused $f+1$ copies;
if (all $f+1$ responses identical)
  use the response;
else invoke recovery algorithm;

Fused Copies:
On receiving updates from unfused copies,
if (all $f+1$ updates identical)
  carry out the update
else invoke recovery algorithm;

Recovery Algorithm:
Accquire all available data structures;
Let t be the number of mismatched primaries;
while $(t > 1)$ do
  choose a copy of some primary X_i with largest multiplicity;
  restart unfused copies of X_i with the state of the chosen copy;
  $t = t - 1$;
endwhile;
// Can assume that t equals one.
// Let X_c be the mismatched primary.
Locate faulty copy among unfused copies of X_c using the locate algorithm;

Locate Algorithm (X_c):
Z: set of unfused copies of X_c $\}$;
Discard copies in Z and fused backups with wrong index/aux structures;
while (there are mismatched copies in Z)
  $w = min\{r : \exists p, q \in Z : value_p[r] \neq value_q[r]\}$;
  Y: $state[w]$ for each copy in Z;
  $j = 1$;
  while (no value in Y with multiplicity $f+1$)
    create, v=$state[w]$ using F_j and all $X_i, i \neq c$ and add v to Y;
    $j = j + 1$;
  endwhile;
  delete copies from Z in which $state[w]! = v$;
endwhile;

Fig. 6. Detection and correction of byzantine faults.

Theorem 7. *Let there be n primaries, each with $O(m)$ nodes of $O(s)$ size each. There exists an algorithm with additional $nf + f$ data structures that can correct f Byzantine faults and has the same overhead as the replication-based approach during normal operation and $O(mfst^2 + nst^2)$ overhead during recovery, where t is the actual number of faults that occurred in the system.*

Proof. We present an algorithm in Fig. 6 that corrects f Byzantine faults. We keep f copies for each primary and f fused data structures overall. This results in additional $nf + f$ data structures in the system. If there are no faults among the unfused copies, all $f+1$ copies will result in the same output and therefore the system will incur the same overhead as the replication-based approach. If the client or one of the fused backups detects a mismatch among the values received from the unfused copies, then the recovery algorithm is invoked. The recovery algorithm first reduces the number of mismatched primaries to one and then uses the locate algorithm to identify the correct primary. We describe the algorithm in greater detail and prove its correctness in the following paragraphs.

The recovery algorithm first checks the number of primaries that are mismatched. First consider the case when there is a single mismatched primary, say X_c. Now given the state of all other primaries, we can successively retrieve the state of X_c from fused data structures $F_j, j = 1 \cdots f$ till we find a copy of X_c that has $f+1$ multiplicity. Now consider the case when there is a mismatch for at least two primaries, say X_c and X_d. Let $\alpha(c)$ and $\alpha(d)$ be the largest multiplicity among unfused copies of X_c and X_d, respectively. Without loss of generality, assume that $\alpha(c) \geq \alpha(d)$. We show that the copy with multiplicity $\alpha(c)$ is correct.

If this copy is not correct, then there are at least $\alpha(c)$ liars among unfused copies of X_c. We now claim that there are at least $f + 1 - \alpha(d)$ liars among unfused copies of X_d which gives us the total number of liars as $\alpha(c) + f + 1 - \alpha(d) \geq f + 1$ contradicting the assumption on the maximum number of liars. Consider the copy among unfused copies of X_d with multiplicity $\alpha(d)$. If this copy is correct we have $f + 1 - \alpha(d)$ liars. If this copy is incorrect, we know that the correct value has multiplicity less than or equal to $\alpha(d)$ and therefore there are at least $f + 1 - \alpha(d)$ liars among unfused copies of X_d. Hence, the primary with multiplicity $\alpha(c)$ is correct. By identifying the correct primary, we have reduced the number of mismatched primaries by 1. By repeating this argument, we get to the case when there is exactly one mismatched primary, say X_c.

We use the locate algorithm in Fig. 6 to locate the correct copy of X_c. In the locate algorithm, we first identify errors in the auxiliary and index structures. Since this information is replicated at all the f fused backups, we can obtain $2f + 1$ versions of this information among which at least $f + 1$ versions are identical (at most f liars). The remaining f versions are certainly faulty and unfused copies with this information can be discarded. This operation can be performed in $O(mf)$ time, as the auxiliary/index structures contain $O(m)$ pointers. If there are no errors among the auxiliary/index structures, we identify errors in the data elements.

The set Z maintains the invariant that it includes all the correct unfused copies (and may include incorrect copies as well). The invariant is initially true because all indices from $1, \ldots, f + 1$ are in Z. Since the set has $f + 1$ indices and there are at most f faults, we know that the set Z always contains at least one correct copy.

The outer *while* loop iterates until all copies are identical. If all copies in Z are identical, from the invariant it follows that all of them must be correct and we can simply return any of the copies in Z. Otherwise, there exist at least two different copies in Z, say p and q. Let w be the first key in which states of copies p and q differ. Either copy p or the copy q (or both) are liars. We now use the fused data structures to recreate copies of $state[w]$, the value associated with key w. Since we have the correct copies of all other primaries $X_i, i \neq c$, we can use them with the fused backups $F_j, j = 1 \cdots f$. Note that

the fused backups may themselves be wrong so it is necessary to get enough multiplicity for any value to determine if some copy is faulty. Suppose that for some v, we get multiplicity of $f+1$. This implies that any copy with $state[w] \neq v$ must be faulty and therefore can safely be deleted from Z. We are guaranteed to get a value with multiplicity $f+1$ out of total $2f+1$ values, viz. $f+1$ values from unfused copies of X_c and f values decoded using the f fused backups and remaining correct primaries. Further, since copies p and q differ in $state[w]$, we are guaranteed to delete at least one of them in each iteration of the inner while loop. Eventually, the set Z would either be singleton or will contain only identical copies, which implies that we have located a correct copy.

The time complexity of reducing the mismatched primaries to one is $O(msft^2)$.We now analyze the time complexity of the procedure *locate*. Assume that there are $t \leq f$ actual faults that occurred. We delete at least one unfused copy of X_c in each iteration of the outer *while* loop and there are at most t faulty data structures giving us the bound of t for the number of iterations of the while loop. In each iteration, creating $state[w]$ requires at most $O(s)$ state to be decoded at each fused data structure at the cost of $O(ns)$. The maximum number of fused data structures that would be required is t. Thus, $O(nts)$ work is required for a single iteration before a copy is deleted from Z. To determine w in incremental fashion requires $O(mfs)$ work cumulative over all iterations. Combining these costs we get the complexity of the algorithm to be $O(mfst^2 + nst^2)$. □

Theorem 7 combines advantages of replication and coding theory. We have enough replication to guarantee that there is at least one correct copy at all times and therefore we do not need to decode the entire state data structure but only locate the correct copy. We have also taken advantage of coding theory to reduce the number of copies from $2f$ to f. It can be seen that our algorithm is optimal in the number of unfused and fused backups it maintains to guarantee that there is at least one correct unfused copy and that faults of any f data structures can be tolerated. The first requirement dictates that there be at least $f+1$ unfused copies and the recovery from Byzantine fault requires that there be at least $2f+1$ fused or unfused copies in all.

6 Practical Example: Amazon's Dynamo

In this section, we present a practical application of our technique based on a real world implementation of a distributed system. Amazon's Dynamo [4] is a distributed data store that needs to provide both durability and very low response times (availability) for writes to the end user. They achieve this using a replication-based solution which is simple to maintain but expensive in terms of space. We propose an alternate design using a combination of both fusion and replication, which consumes far less space, while guaranteeing nearly the same levels of durability and availability.

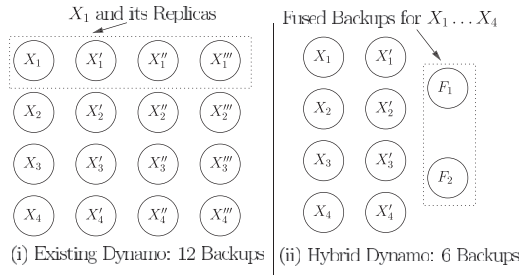


Fig. 7. Design strategies for dynamo.

6.1 Existing Dynamo Design

We present a simplified version of Dynamo with a focus on the replication strategy. Dynamo consists of clusters of primary hosts each containing a data store like a hash table that stores key-value pairs. The key space is partitioned across these hosts to ensure sufficient load-balancing. For both fault tolerance and availability, f additional copies of each primary hash table are maintained. These $f+1$ identical copies can correct f crash faults among the primaries. The system also defines two parameters r and w which denote the minimum number of copies that must participate in each read request and write request respectively. These values are each chosen to be less than f. In Fig. 7(i), we illustrate a simple set up of Dynamo for $n=4$ primaries, with $(f,w,r)=(3,2,2)$.

To read and write from the data store, the client can send its request to any one of the $f+1$ copies responsible for the key of the request, and designate it as the *coordinator*. The coordinator reads/writes the value corresponding to the key locally and sends the request to the remaining f copies. On receiving $r-1$ or $w-1$ responses from the backup copies for read and write requests, respectively, the coordinator responds to the client with the data value (for reads) or just an acknowledgment (for writes). Since $w < f$, clearly some of the copies may not be up to date when the coordinator responds to the client. This necessitates some form of data versioning, and the coordinator or the client has to reconcile the different data versions on every read. This is considered an acceptable cost since Dynamo is mainly concerned with optimizing writes to the store. In this setup, when one or more data structures crash, the remaining copies responsible for the same key space can take over all requests addressed to the failed data structures. Once the crashed data structure comes back, the copy that was acting as proxy just transfers back the keys that were meant for the node. In Fig. 7(i), since there can be at most three crash faults in the system, there is at least one node copy for each primary remaining for recovery.

6.2 Hybrid Dynamo Design

We propose a hybrid design for Dynamo that uses a combination of fusion and replication. We focus on the case of $(f,w,r)=(3,2,2)$. Instead of maintaining three additional copies for each primary ($f=3$), we maintain just a single additional copy for each primary and two fused backups for the entire set of primaries as shown in Fig. 7(ii). The fused backups achieve the savings in space while the additional copies allows the necessary availability for reads. The fused backups along with the additional copies can correct three crash faults among the primaries. The basic

protocol for reads and writes remains the same except for the fact that the fused backups cannot directly respond to the client requests since they require the old value associated with the key (Section 3). On receiving a write request, the coordinator can send the request to these fused backups which can respond to the request after updating the table. For the case of $w = 2$, as long as the coordinator, say X_i obtains a response from one among the three backups (one copy and two fused backups) the write can succeed. This is similar to the existing design and hence performance for writes is not affected significantly. On the other hand, performance for reads does drop since the fused backups that contain data in the coded form cannot return the data value corresponding to a key in an efficient manner. Hence, the two additional copies need to answer all requests to maintain availability. Since Dynamo is optimized mainly for writes, this may not be a cause for concern. To alleviate the load on the fused backups, we can partition the set of primaries into smaller blocks, trading some of the space efficiency for availability. For the set up shown in Fig. 7, we can maintain four fused backups where F_1, F_2 are the fused backups for X_1 and X_2, while F_3 and F_4 are the fused backups of X_3 and X_4.

Similar to the existing design of Dynamo, when data structures crash, if there are surviving copies responsible for the same keys, then they can take over operation. However, since we maintain only one additional copy per primary, it is possible that none of the copies remain. In this case, the fused backup can *mutate* into one or more of the failed primaries. It can receive requests corresponding to the failed primaries, update its local hash table and maintain data in its normal form (without fusing them). Concurrently, to recover the failed primaries, it can obtain the data values from the remaining copies and decode the values. Hence, even though transiently the fault tolerance of the system is reduced, there is not much reduction in operational performance. Dynamo has been designed to scale to 100 hosts each containing a primary. So in a typical cluster with $n = 100$, $f = 3$ the original approach requires, $n * f = 300$ backup data structures. Consider a hybrid solution that maintains one additional copy for each primary and maintains two fused backups for every 10 primaries. This approach requires only $100 + 20 = 120$ backup data structures. This results in savings in space, as well as power and other resources required by the processes running these data structures. Hence, the hybrid solution can be very beneficial for such a real-world system.

7 Implementation and Results

In this section, we describe our fusion-based data structure library [1] that includes all data structures provided by the Java Collection Framework. Further we have implemented our fused backups using Cauchy RS codes (referred to as *Cauchy-Fusion*) and Vandermonde RS codes (*Van-Fusion*). We refer to either of these implementations as the *current* version of fusion. We have compared its performance against replication and the older version of fusion (*Old-Fusion*) [7]. Old-Fusion has a different, simpler design of the fused backups, similar to the one presented in the design motivation of Section 3. We extend it for f-fault tolerance using Vandermonde RS codes. The current versions of fusion, using either Cauchy or Vandermonde RS, outperform the older version on all three counts: backups space, update time at the backups and time taken for recovery. In terms of comparison with replication, we achieve almost n times savings in space as confirmed by the theoretical results, while not causing too much update overhead. Recovery is much cheaper in replication.

Fault-tolerant data structure library. We implemented fused backups and primary wrappers for the data structures in the Java 6 Collection framework that are broadly divided into list-based, map-based, set-based, and queue-based data structures. We evaluated the performance of a representative data structure in two of these categories: linked lists for list-based and tree maps for map-based data structures. Both Old-Fusion and Van-Fusion use Vandermonde RS codes with field size 2^{16}, while Cauchy-Fusion uses Cauchy RS codes, with field size 2^5. The RS codes we have used are based on the C++ library provided by Plank et al. [15], [16]. Currently, we just support the Integer data type for the data elements at the primaries.

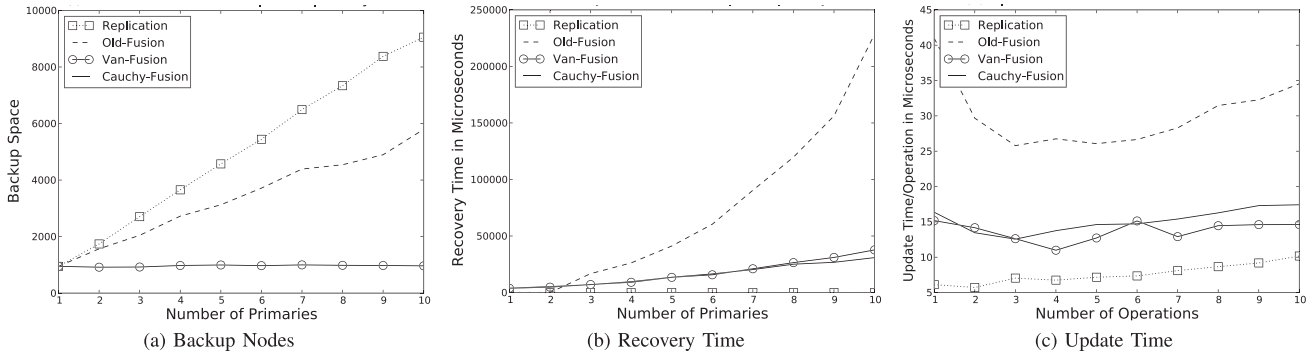
Evaluation. We implemented a distributed system of hosts, each running either a primary or a backup data structure and compared the performance of the four solutions: Replication, Old-Fusion, Van-Fusion, and Cauchy-Fusion. The algorithms were implemented in Java 6 with TCP sockets for communication and the experiments were executed on a single Intel quad-core PC with 2.66 GHz clock frequency and 12 GB RAM. In the future, we wish to evaluate fusion over physically disparate machines. The three parameters that were varied across the experiments were the number of primaries n, number of faults f and the total number of operations performed per primary, ops. The operations were biased toward inserts (80 percent) and the tests were averaged over five runs. In our experiments, we only assume crash faults. We describe the results for the three main tests that we performed for linked lists: backup space, update time at the backup and recovery time (Fig. 8). The results for tree maps are of a similar nature (Fig. 10 in the Appendix, which is available in the online supplemental material,).

7.1 Backup Nodes

To measure the space required by the backups, we assume that the size of data far exceeds the overhead of the index structure and hence, we just plot the total number of backup nodes required by each solution. We fix $f = 3$, $ops = 500$ and vary n from 1 to 10. Cauchy-Fusion and Van-Fusion, differ only in the type of RS code used, but use the same design for the backups. So, they both require the same number of backup nodes. Both Cauchy-Fusion and Van-Fusion perform much better than both replication (approximately n times) and Old-Fusion (approximately $n/2$ times) because the number of nodes per backup never exceeds the maximum among the primaries.

7.2 Recovery Time

We measure recovery time as the time taken to recover the state of the crashed data structures *after* the client obtains the state of the requisite data structures. The same experiment as that used to measure backup space was used to compare the four solutions. Cauchy-Fusion and Van-Fusion perform



(a) Backup Nodes (b) Recovery Time (c) Update Time

Fig. 8. Linked lists: experimental evaluation comparing replication, old-fusion, van-fusion, and cauchy-fusion.

much better than Old-Fusion (approximately $n/2$ times) because recovery in fusion involves iterating through all the nodes of each fused backup. The current design contains fewer nodes and hence performs better. The time taken for recovery by replication is negligible as compared to fusion-based solutions (the curve is almost merged with the x-axis in the graphs). This is to be expected since recovery in replication requires just copying the failed data structures after obtaining them. However, note that, even for $n = 10$, the time taken for recovery by both Cauchy and Van-Fusion is under 40 milliseconds. This can be a small cost to pay for the considerable savings that we achieve in space.

Further analysis of the recovery times in both Cauchy-Fusion and Van-Fusion shows that almost 40 percent of the cost of recovery is spent in decoding the coded data elements. This implies two things. First, using a different code such as LDPC codes, that offers faster decoding in exchange for less space efficiency, fusion can achieve faster recovery times. Second, more than 50 percent of recovery time is spent on just iterating through the backup nodes, to retrieve the data for decoding. Hence, optimizing the recovery algorithm, can reduce the recovery time. The other observation is that, even though Cauchy RS codes have much faster decode times than Vandermonde RS codes, the recovery time for Cauchy-Fusion is only marginally better than Van-Fusion. We believe this is mainly due to the small data size (4-byte integers). For larger data values, Cauchy-Fusion might perform much better than Van-Fusion. These are future areas of research that we wish to explore.

7.3 Update Time

Finally, to measure the update time at the backups, we fixed $n = 3$, $f = 1$ and varied ops from 500 to 5,000. Both Cauchy-Fusion and Van-Fusion has more update overhead as compared to replication (approximately 1.5 times slower) while they perform better than the older version (approximately 2.5 times faster). Since the current design of fused backups has fewer backup nodes, it takes lesser time to iterate through the nodes for an update. The update time at a backup can be divided into two parts: the time taken to locate the node to update plus the time taken to update the node's code value. The code update time was insignificantly low and almost all the update time was spent in locating the node. Hence, optimizing the update algorithm can reduce the total update time considerably. This also explains why Cauchy-Fusion does not achieve any improvement over Van-Fusion and at times does slightly worse, because the overhead of dealing with blocks of data in Cauchy-Fusion exceeds the savings achieved by faster updates. As mentioned before, we believe that with the larger data sizes, Cauchy-Fusion may perform as expected.

8 Comparative Study: Replication versus Fusion

In this section, we summarize the main differences between replication and fusion (Table 1). Throughout this section, we assume n primary data structures, containing at most $O(m)$ nodes of size $O(s)$ each. Each primary can be updated in $O(p)$ time. We assume that the system can correct either f crash faults or f Byzantine faults, and t is the actual number of faults that occur. Note that, the comparison in this section is independent of the type of data structure used. We assume that the fusion operator is RS coding, which only requires f parity blocks to correct f erasures among a given set of data blocks.

8.1 Number of Backups

To correct f crash faults among n primaries, fusion requires f backup data structures as compared to the nf backup data structures required by replication. For Byzantine faults, fusion requires $nf + f$ backups as compared to the $2nf$ backups required by replication.

8.2 Backup Space

For crash faults, the total space occupied by the fused backups is msf (f backups of size ms each) as compared to $nmsf$ for replication (nf backups of size ms each). For Byzantine faults, since we maintain f copies of each primary along with f fused backups, the space complexity for fusion is $nfms + msf$ as compared to $2nmsf$ for replication.

8.3 Maximum Load on Any Backup

We define load as the number of primaries each backup has to service. Since each fused backup has to receive requests from all n primaries the maximum load on the fused backup is n times more than the load for replication. Note that,

TABLE 1
Replication versus Fusion

	Rep-Crash	Fusion-Crash	Rep-Byz	Fusion-Byz
Number of Backups	nf	f	$2nf$	$nf + f$
Backup Space	$nmsf$	msf	$2nmsf$	$nmsf + msf$
Max Load/Backup	$O(1)$	$O(n)$	$O(1)$	$O(n)$
Normal Operation Time	$O(p)$	$O(p)$	$O(p)$	$O(p)$
Recovery Time	$O(mst)$	$O(mst^2n)$	$O(msf)$	$O(msft^2 + nsi^2)$
Normal Operation Messages	f msgs, size s each	f msgs, size $2s$ each	$2f$ msgs, size s each	f msgs size s, f msgs size $2s$
Recovery Messages	t msgs, size ms each	$n+f-t$ msgs, size ms each	$2t + 1$ msgs, size ms each	$nf + n + f$ msgs, size ms each

higher the value of n more the savings in space/number of backups ($O(n)$ times), but more the maximum load on any backup (again, $O(n)$ times).

8.4 Normal (Fault-Free) Operation Time

The fused backups in our system can be updated with the same time complexity as that for updating the corresponding primary, i.e., $O(p)$. We have shown that the updates at the backup can be received in any order and hence, there is no need for synchrony. Also, if Byzantine faults/liars need to be detected with every update in a system, then fusion causes no overhead in time.

8.5 Recovery Time

This parameter refers to the time complexity of recovery at the client, after it has acquired the state of the relevant data structures. In the case of fusion, to recover from t ($t \leq f$) crash faults, we need to decode the backups with total time complexity $O(mst^2n)$. For replication, this is only $O(mst)$. For Byzantine faults, fusion takes $O(mfst^2 + nst^2)$ to correct t Byzantine faults. In the case of replication this is only $O(msf)$. Thus, replication is much more efficient than fusion in terms of the time taken for recovery. However, since we assume faults to be rare, the cost of recovery may be acceptable.

8.6 Normal (Fault-Free) Operation Messages

This parameter refers to the number of messages that the primary needs to send to the backups for any update. We assume that the size of the key for insert or delete is insignificantly small as compared to the data values. In fusion, for crash faults, every update sent to the primary needs to be sent to f backups. The size of each message is $2s$ since we need to send the new value and old value to the backups. For deletes, the size of each message is $2s$ since we need to send the old value and the value of the top-of-stack element (as shown in Fig. 5). Hence, for crash faults, in fusion, for any update, f messages of size $2s$ need to be exchanged. For replication, in inserts, only the new value needs to be sent to the f copies of the primary and for deletes, only the key to be deleted needs to be sent. Hence, for crash faults in replication, for any update f messages of size at most s need to be exchanged.

For Byzantine faults, for fusion, since we maintain f copies of each primary and f fused backups, it needs f messages of size s and f messages of size $2s$, respectively. In replication, $2f$ messages of size s need to be sent to the $2f$ copies of the primary for inserts and for deletes, only $2f$ keys need to be sent.

8.7 Recovery Messages

This refers to the number of messages that need to be exchanged once a fault has been detected. When t crash faults are detected, in fusion, the client needs to acquire the state of all the remaining data structures. This requires $n + f - t$ messages of size $O(ms)$ each. In replication the client only needs to acquire the state of the failed copies requiring only t messages of size $O(ms)$ each. For Byzantine faults, in fusion, the state of all $n + nf + f$ data structures (primaries and backups) needs to be acquired. This requires $nf + f$ messages of size $O(ms)$ each. In replication, only the state of any $2t + 1$ copies of the faulty primary are needed, requiring just $2t + 1$ messages of size $O(ms)$ each.

9 Conclusion

Given n primaries, we present a fusion-based technique for fault tolerance that guarantees $O(n)$ savings in space as compared to replication with almost no overhead during normal operation. We provide a generic design of fused backups and their implementation for all the data structures in the Java Collection framework that includes vectors, stacks, maps, trees, and most other commonly used data structures. We compare the main features of our work with replication, both theoretically and experimentally. Our evaluation confirms that fusion is extremely space efficient while replication is efficient in terms of recovery, load on the backups and the size of the messages that need to be sent to the backups. We wish to explore alternate techniques for fusion with a focus on erasure codes such as LDPC codes [5] and LT codes [10] that offer different tradeoffs between the various system parameters.

Many real world systems such Amazon's Dynamo or Google's MapReduce framework use replication extensively for fault tolerance. Using concepts presented in this paper, we can consider an alternate design using a combination of replication and fusion-based techniques. We illustrate this in Section 6 by presenting a simple design alternative for Amazon's data store, Dynamo. In a typical Dynamo cluster of 100 hosts our solution requires only 120 backup structures as compared to the existing set up of 300 backup structures, without compromising on other important QoS parameters such as response times. Thus fusion achieves significant savings in space, power, and other resources.

Acknowledgments

This research was supported in part by the US National Science Foundation (NSF) Grants CNS-0718990, CNS-0509024, Texas Education Board Grant 781, SRC Grant 2006-TJ-1426, and Cullen Trust for Higher Education Endowed Professorship.

References

[1] B. Balasubramanian and V.K. Garg, "Fused Data Structure Library (Implemented in Java 1.6)," Parallel and Distributed Systems Laboratory, http://maple.ece.utexas.edu, 2010.
[2] E.R. Berlekamp, *Algebraic Coding Theory.* McGraw-Hill, 1968.
[3] J. Blömer, M. Kalfane, M. Karpinski, R. Karp, M. Luby, and D. Zuckerman, "An XOR-Based Erasure-Resilient Coding Scheme," Technical Report TR-95-048, Int'l Computer Science Inst., Aug. 1995.
[4] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's Highly Available Key-Value Store," *Proc. 21st ACM SIGOPS Symp. Operating Systems Principles (SOSP '07),* pp. 205-220, 2007.
[5] R. Gallager, "Low-Density Parity-Check Codes," *IRE Trans. Information Theory,* vol. IT-8, no. 1, pp. 21-28, Jan. 1962.
[6] V.K. Garg, "Implementing Fault-Tolerant Services Using State Machines: Beyond Replication," *Proc. 24th Int'l Conf. Distributed Computing (DISC),* pp. 450-464, 2010.
[7] V.K. Garg and V. Ogale, "Fusible Data Structures for Fault Tolerance," *Proc. 27th Int'l Conf. Distributed Computing Systems (ICDCS '07),* June 2007.
[8] L. Lamport, "The Implementation of Reliable Distributed Multiprocess Systems," *Computer Networks,* vol. 2, pp. 95-114, 1978.

[9] L. Lamport, R.E. Shostak, and M.C. Pease, "The Byzantine Generals Problem," *ACM Trans. Programming Languages and Systems,* vol. 4, no. 3, pp. 382-401, 1982.
[10] M. Luby, "LT Codes," *Proc. 43rd Symp. Foundations of Computer Science (FOCS '02),* p. 271, 2002.
[11] V. Ogale, B. Balasubramanian, and V.K. Garg, "A Fusion-Based Approach for Tolerating Faults in Finite State Machines," *Proc. IEEE Int'l Symp. Parallel and Distributed Processing (IPDPS '09),* pp. 1-11, 2009.
[12] J.K. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, M. Rosenblum, S.M. Rumble, E. Stratmann, and R. Stutsman, "The Case for RAMClouds: Scalable High-Performance Storage Entirely in Dram," *ACM SIGOPS Operating Systems Rev.,* vol. 43, pp. 92-105, 2009.
[13] D.A. Patterson, G. Gibson, and R.H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '88),* pp. 109-116, 1988.
[14] W.W. Peterson and E.J. Weldon, *Error-Correcting Codes - Revised,* second ed. The MIT Press, Mar. 1972.
[15] J.S. Plank, "A Tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-Like Systems," *Software - Practice and Experience,* vol. 27, no. 9, pp. 995-1012, Sept. 1997.
[16] J.S. Plank, S. Simmerman, and C.D. Schuman, "Jerasure: A Library in C/C++ Facilitating Erasure Coding for Storage Applications - Version 1.2," Technical Report CS-08-627, Univ. of Tennessee, Aug. 2008.
[17] J.S. Plank and L. Xu, "Optimizing Cauchy Reed-Solomon Codes for Fault-Tolerant Network Storage Applications," *Proc. IEEE Fifth Int'l Symp. Network Computing and Applications,* pp. 173-180, 2006.
[18] M.O. Rabin, "Efficient Dispersal of Information for Security, Load Balancing, and Fault Tolerance," *J. ACM,* vol. 36, no. 2, pp. 335-348, 1989.
[19] I.S. Reed and G. Solomon, "Polynomial Codes over Certain Finite Fields," *J. Soc. for Industrial and Applied Math.,* vol. 8, no. 2, pp. 300-304, 1960.
[20] F.B. Schneider, "Byzantine Generals in Action: Implementing Fail-Stop Processors," *ACM Trans. Computer Systems,* vol. 2, no. 2, pp. 145-154, 1984.
[21] F.B. Schneider, "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial," *ACM Computing Surveys,* vol. 22, no. 4, pp. 299-319, 1990.
[22] C.E. Shannon, "A Mathematical Theory of Communication," *Bell Systems Technical J.,* vol. 27, pp. 379-423 and 623-656, 1948.
[23] B. Balasubramanian and V.K. Garg, "Fused Data Structures for Handling Multiple Faults in Distributed Systems," *Proc. Int'l Conf. Distributed Computing Systems (ICDCS '11),* pp. 677-688, June 2011.

Bharath Balasubramanian received the BE degree in electronics from Mumbai University in 2004, and the MS and PhD degrees in computer engineering from the University of Texas at Austin in 2007 and 2012, respectively. Currently, he is a postdoctoral research associate in the electrical engineering department at Princeton University with Dr. Mung Chiang. His areas of interest include: concurrent and distributed algorithms, fault tolerant distributed systems, distributed storage, and distributed debugging.

Vijay K. Garg received the bachleor's degree from the Indian Institute of Technology, Kanpur, India, in 1984 and the MS and PhD degrees from the University of California, Berkeley, in 1985 and 1988, respectively. He is a Cullen Trust Endowed professor in the Department of Electrical and Computer Engineering and the director of the Parallel and Distributed Systems Laboratory at The University of Texas at Austin. His main contributions include the areas of global predicate detection, distributed debugging, distributed simulation, fault-tolerance, distributed algorithms, and supervisory control of discrete event systems. He is the author of the books *Concurrent and Distributed Computing in Java* (Wiley and Sons), *Elements of Distributed Computing* (Wiley and Sons), *Principles of Distributed Systems* (Springer-Verlag) and a coauthor of the book *Modeling and Control of Logical Discrete Event Systems* (Springer-Verlag). He is a fellow of the IEEE.