Implementing Fault-Tolerant Services Using State Machines: Beyond Replication

Vijay K. Garg *

Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX 78712-1084, USA

Abstract. This paper describes a method to implement fault-tolerant services in distributed systems based on the idea of fused state machines. The theory of fused state machines uses a combination of coding theory and replication to ensure efficiency as well as savings in storage and messages during normal operations. Fused state machines may incur higher overhead during recovery from crash or Byzantine faults, but that may be acceptable if the probability of fault is low. Assuming n different state machines, pure replication based schemes require $n(f+1)$ replicas to tolerate f crash faults in a system and $n(2f+1)$ replicas to tolerate f Byzantine faults. For crash faults, we give an algorithm that requires the optimal f backup state machines for tolerating f faults in the system of n machines. For Byzantine faults, we propose an algorithm that requires only $nf+f$ additional state machines, as opposed to $2nf$ state machines. Our algorithm combines ideas from coding theory with replication to provide low overhead during normal operation while keeping the number of copies required to tolerate f faults small.

1 Introduction

The replicated state machine approach is a general method for implementing a fault-tolerant service by replicating servers and coordinating client interactions with server replicas. This approach proposed by Lamport in [1, 2] and further elaborated by Schneider in [3] is considered the standard solution to the problem of fault-tolerance in distributed systems. Note that replication has been considered wasteful in the context of fault-tolerance of data (in communication and storage) for many decades, but in the distributed systems replication continues to be the dominant approach for fault-tolerance [4]. In this paper, we give an alternate method for fault-tolerance that combines ideas from replication with coding theory [5, 6] to get main advantages of both the approaches. We use (sufficient) replication to guarantee low overhead during normal operations and coding theory to reduce the number of copies to get space and message savings.

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

We depart from the standard model of fault-tolerance in distributed systems in which the problem is to tolerate faults in functioning of a single state machine. We will be concerned with fault-tolerance in a *set* of state machines where the size of the set will usually be greater than one. While this assumption makes the problem different from the usual set-up, we argue that our set-up is practically useful. Any large system is generally constructed as a set of state machines rather than a single monolithic state machine. Even when the server is constructed as a single state machine, it is quite natural to have multiple instances of the state machines deployed for different departments of the organization.

In this paper, we show how services in a distributed system can be made fault-tolerant using fusion. Given n *different* state machines running on different servers, we focus on tolerating f faults. We focus on two types of faults: *crash* faults and *Byzantine* faults. For crash faults, faulty state machines lose their state. We assume that crash faults are detectable and the problem that remains is to recover the lost state of state machines. For Byzantine faults [7], the state machine may go to an incorrect state spontaneously and the algorithm must continue to provide correct responses to the client in spite of these faults.

For *crash* faults, we give a technique to construct additional f state machines (called fused state machines) such that the system of $n + f$ machines can tolerate crash of any f machines in the system. We illustrate our technique on the resource allocation service from [3], a causal ordering algorithm [8] and a distributed mutual exclusion algorithm [9]. The fused state machines use a combination of erasure coding and replication to ensure that during normal operations, the message and computation overhead on primary state machines is close to that for replicated state machines. The updates of fused state machines are made efficient using linearity of erasure coding scheme employed and sufficient replication.

For *Byzantine* faults, the problem of detection is harder from the perspective of computation and communication complexity. Here we use a hybrid of replication and coding theory. In particular, we give an algorithm that keeps the overhead of the replicated state machine approach during normal operations but requires only $nf + f$ additional state machines (as opposed to $2nf$ state machines). Our algorithm is based on the following observation that if there are $f + 1$ copies of a state machine, then at least one of them is correct. In case of a fault, we only need to determine which of these copies is correct. The traditional method of keeping $2f + 1$ copies (and then using majority) is wasteful for the task. We introduce the notion of *efficient liar detection* based on fused state machines. This allows us to prove the following main result in this paper (in Section 3). Let there be n primary state machines, each with $O(m)$ data structures. There exists an algorithm with additional $nf + f$ state machines that can tolerate f Byzantine faults and has the same overhead as the Replicated State Machine approach during the normal operation and additional $O(mf + nt^2)$ overhead during recovery where t is the actual number of faults that occurred in the system.

In our earlier work, we have given algorithms for fusible data structures. In particular, [10] gives algorithms for arrays, stacks, queues, linked lists etc. to handle *crash* faults. This work has been generalized to tolerate multiple *crash* faults in [11]. In contrast, the goal of the current work is to focus on the differences between the replicated state machine approach and the fused state machine approach and also tackle *Byzantine* faults. Furthermore, we show that our approach is applicable to many distributed algorithms including a causal ordering algorithm [8], and Ricart and Agrawala's mutual exclusion algorithm[9]. For both of these algorithms, we get n-fold savings in space. We also get savings in messages for Ricart and Agarwala's algorithm because of aggregation that is possible in fused state machines. In [12], an algorithm has been provided to generate fused *finite* state machines. That algorithm assumes that the state space of the primary machines is finite. In this paper, techniques are suitable even for infinite state space.

In data storage and communication, coding theory is extensively used to recover from faults. For example, RAID disks use disk striping and parity based schemes (or erasure codes) to recover from the disk faults [13–15]. As another example, network coding [16, 17] has been used for recovering from packet loss or to reduce communication overhead for multicast. In these applications, the data is viewed as a set of data entities such as disk blocks for storage applications and packets for network applications. Coding theory techniques [6] are oblivious to the structure of the data. The details of actual operations on the data are ignored and the codes are simply recomputed after any write update. To tolerate crash failures for servers, one can view the memory of the server as a set of pages and apply coding theory to maintain code words. This approach, however, may not be practical because a small change in data may require recomputation of the backup for one or more pages. This results in a high computational and communication overhead. We show in this paper that with data structure-aware programming and partial state replication, backup machines can be designed so that they provide fault-tolerance in an efficient manner.

2 Fusible State Machines

There are n *deterministic* primary state machines, $P(i)$, where i ranges from 1 to n. Each state machine receives an input from the client (or environment). On receiving the input, the state machine applies the state transition function to change its state. The set of states and inputs may be infinite.

We require state machines to be deterministic just as required by the replicated state machine approach. Given the state of a machine and the sequence of inputs, the behavior of the state machine is required to be unique. This assumption is crucial in both the replicated state machine (RSM) and the fused state machine (fused-SM) approaches.

Throughout this paper we assume that channels are reliable and FIFO and that there is a fixed upper bound for all message delivery. We also assume that crashes of processes are reliably detected.

2.1 Event Counter

To concretize our discussion, we start with n simple state machines, $P(i)$'s, shown in Fig. 1. Each of these n machines accept two types of input: $entry(v)$ and $exit(v)$. These state machines may, for example, be counting the number of people of type i entering a room. Each state machine has a variable *count* with domain as non-negative integers. When $P(i)$ receives an event $entry(v)$, it increments its count if v is equal to i and decrements it when it receives a similar $exit(v)$ event.

$P(i) :: i = 1..n$ int $count_i = 0$; On event $entry(v)$: **if** $(v == i)$ $count_i = count_i + 1$; On event $exit(v)$: **if** $(v == i)$ $count_i = count_i - 1$;	$F(j) :: j = 1..f$ int $fCount_j = 0$; On event $entry(i)$, for any i $fCount_j = fCount_j + i^{j-1}$; On event $exit(i)$ for any i $fCount_j = fCount_j - i^{j-1}$;

Fig. 1. Fusion of Counter State Machines

To tolearte f faults, the Replicated State Machine (RSM) approach requires f additional state machines for each of $P(i)$ resulting in the total of nf additional state machines. For fusion, we add just f additional machines, $F(1)..F(f)$ as shown in Fig. 1. $F(j)$ increases its count by i^{j-1} for any event $entry(i)$ and decrements by the same amount for $exit(i)$. It can be seen that the fused-SM $F(1)$ tracks the sum of all counts. It increments the variable $fCount_1$ on $entry(i)$ for any i and decrements it for any $exit(i)$. $F(2)$ maintains $fCount_2 = \sum_i i * count_i$. More generally, $fCount_j = \sum_i i^{j-1} * count_i \quad for\ all\ j = 1..f$.

The recovery procedure for fusible SM is more complex than for replication. It crucially depends on the fact that if any f machines crash, the rest of the machines are still available. $F(1)$ is sufficient to recover from one crash fault. If $P(c)$ has failed, then its state $count_c$ can be recovered as $fCount_1 - \sum_{i \neq c} count_i$. In general, we can recover states of any f failed state machines using the remaining machines. For example, consider the case when f is two and the machines that crashed are $P(c)$ and $P(d)$. Using fusion machine $F(1)$ and remaining counts we can get the value of $count_c + count_d$. Using $fCount_2$, we can get the value of $c * count_c + d * count_d$. We have two linearly independent equations in two variables which can be solved to get the values of $count_c$ and $count_d$. More generally, recovery from f faults reduces to solving f linearly independent equations in f variables.

A reader well-versed in coding theory would realize that if $(count_1, count_2, , count_n)$ is viewed as data, ($count_1$,$count_2$,..$count_n$,$fCount_1$, $fCount_2$,..$fCount_f$) can be viewed as a code word. The code word obtained is equivalent to one obtained by multiplying data vector by the identity matrix adjoined with the transpose

of the Vandermonde matrix[5]. The unique solvability of all the counts is easy to show; the proof is given in [18] for completeness sake.

Theorem 1. *Suppose* $\mathbf{x} = (count_1, count_2, , count_n)$ *is the state of the primary state machines. Assume* $fCount_j = \sum_i i^{j-1} * count_i$ *for all* $j = 1..f$. *Given any* n *values out of* $\mathbf{y} = (count_1, count_2, ..count_n, fCount_1, fCount_2, ..fCount_f)$ *the remaining values in* $\mathbf{x}$ *can be uniquely determined.*

It is important to note the distinction between a server and a state machine. In the event counter example, to tolerate f crash faults among n state machines, the RSM approach need not run all nf on distinct servers. We could, for example, run one copy of each of the state machines for all $P(1)..P(n)$, on one server. Thus, the number of servers required to tolerate f crash faults can still be considered to be f for the RSM approach. However, the fused state machine approach provides upto n-fold savings in the space required for keeping backups. We now show that the fused-SM approach also yields benefits in computation and communication when events are shared between primary state machines. Suppose that each $P(i)$ has an additional event called *incr* which increases $count_i$ by 1. When the event *incr* happens, all P_i increment their counts. In the RSM approach the event *incr* would be communicated to nf state machines, and will be executed nf times. In the Fused-SM approach, we require $F(j)$ to execute *incr* as $fCount_j = fCount_j + \sum_{i=1}^{i=n} i^{j-1}$. The total number of events that are executed is exactly f, one for each fused-SM. Thus, when events are shared across primary state machines, we get the advantage of *aggregation* thereby reducing the message and computation complexity for backup.

Note that we do not require the fused-SMs to be synchronized with primary state machines. The only requirement is that all updates from primary state machines are applied in the same order at all the fused-SMs. The messages at fused-SMs may be buffered because the primary state machines never wait for fused-SMs to finish their updates. In case of a failure of a primary machine, all the pending updates at the fused-SMs must be applied before the recovery.

So far we had assumed that by adding numbers we do not get overflow. If overflow is possible, there are two approaches to tackle it. The first approach is to do all the arithmetic, i.e. addition (subtraction), and multiplication (division) in finite Galois field as typically done in coding theory [5]. In that case the matrix G can either be chosen as a Cauchy Matrix or a Vandermonde matrix reduced using elementary transformations so that the first n rows form an identity matrix [19]. The other possibility is to guarantee that there is never any overflow in any computation. This can be done, for example, by using `BigInteger` package in Java.

2.2 Causal Ordering

We now generalize the Event Counter example to primary state machines that contain not one variable but a set of data structures. Whenever a primary state machine receives an event from a client and updates it data structures, it also

sends a message to the fused state machines with the list of variables and the incremental change in their values. We illustrate this method for a *causal ordering* algorithm [20] in a group of n processes.

Consider the version described by Raynal, Schiper, and Toueg [8]. Each process maintains a matrix M of integers. The entry $M[q, r]$ at $P(i)$ records the number of messages sent by process $P(q)$ to process $P(r)$ as known by process $P(i)$. Whenever a message is sent from $P(i)$ to $P(r)$, the matrix M is piggybacked with the message. A message is eligible to be received when the number of messages sent from any process $P(q)$ to $P(i)$, as indicated by the matrix W received in the message, is less than or equal to the number recorded in the matrix M.

Suppose, we would like the system to be able to tolerate f crash faults, i.e., recover matrices for processes that have crashed. We require $P(i)$'s to send an "M-Update" message with incremental changes in entries of the matrix to the fused processes $F(j)$. Instead of maintaining f copies of the matrix for each primary process, the fused-SM algorithm requires a single (fused) matrix for every fault. Thus, the storage requirement for fused processes is $O(fn^2)$ as opposed to $O(fn^3)$ required by a replication based algorithm. A similar algorithm can be used to recover vector clocks of faulty processes in distributed systems.

2.3 Resource Allocator

The technique outlined in previous section may not be practical when a simple change in data structure results in a significant change in the state. We show that by analysis of the data structure, and by selective replication the size of the messages from primary messages to fusion processes can be reduced significantly.

To illustrate this point, we apply the method of fusion to the resource allocator state machine in [3]. Assume that there are n different type of resources that can only be used in mutually exclusive fashion. The state machine $P(i)$ shown in Fig. 2 handles clients requesting resource i. It maintains two variables: *user*, an integer which records the current user of the resource if any, and *waiting*, a queue of integers which stores the id's of clients waiting for the resource.

user: int initially 0;// resource idle	**On receiving** release
waiting: queue of int initially null;	if (*waiting*.isEmpty())
On receiving acquire from client *pid*	user = 0;
if (*user* == 0) {	else { user = *waiting*.head();
send(OK) to client *pid*; *user* = *pid*;}	send(OK) to *user*;
else *waiting*.append(*pid*);	*waiting*.removeHead(); }

Fig. 2. Resource Allocator State Machine from [3] $P(i) :: i = 1..n$

Suppose that we want to tolerate one fault in any of these n machines. Whenever, the variable *user* changes we can send the incremental change to fusion

processes. But, what should we do about the waiting list? If we view the bit representation of waiting list as an integer (a big integer), then computing the code at fusion processes after every change would be very inefficient. We use the technique from fusible data structures[10]. Instead of sending the change in state, we send the event that allows the fused structure to be maintained efficiently. The primary state machine that uses fused-SM approach is shown in Fig. 3. Whenever any data structure changes, it sends to the fused machines the change that needs to be made in the data structure in a manner that is tailored to the data structure. Note that the primary machine does not send the changed queue or even the incremental difference from the old queue and the new queue. It only sends enough information so that the fused queues can carry out the state change.

$P(i) :: i = 1..n$
On receiving acquire from client *pid*
if ($user == 0$){send(OK) to client *pid*;
    $user = pid$;
    send(USER, i, $user$) to $F(j)$'s;}
else { $waiting$.append(pid);
    send(ADD-WAITING,i, pid) to $F(j)$'s;}

On receiving release
if (waiting.isEmpty()){$olduser = user$;
    $user = 0$;
    send(USER, i, $user - olduser$) to $F(j)$ }
else { $olduser = user$;
    $user = waiting.head()$;
    send(OK) to $waiting.head()$;
    $waiting.removeHead()$;
    send(USER, i, $user - olduser$) to $F(j)$'s
    send(DEL-WAITING, i, $user$) to $F(j)$'s
}

$F(j) :: j = 1..f$
    $fuser$:int initially 0;
    $fwaiting$:fused queue initially 0;

On receiving (USER, i, val)
$fuser = fuser + i^{j-1} * val$;

On receiving (ADD-WAITING, i, pid)
$fwaiting.append(i, pid)$;

On receiving (DEL-WAITING, i, $user$)
$fwaiting.deleteHead(i, user)$;

Fig. 3. Algorithm A: Fused State Machine for Resource Allocation

The code for the fused-SM is shown in Fig. 3. In $F(j)$ we have used $fwaiting$ as a fused queue. For simplicity, we use a circular array based implementation (a linked list based implementation is in [10]).

The above method has reduced the number of backup state machines nf to f and yet it can tolerate any f faults from $P(1)..P(n)$. The recovery process is more complex than replication but the significant savings (n-fold) in the reduced number of active state machines may justify this added complexity especially when the probability of faults is small.

Remark: So far we had assumed that the clients interact only with the primary machines which, in turn, interacted with fusion machines to keep them up-to-date. In many examples, an alternate design is possible in which the com-

```
fQueue: array[0..M − 1] of int initially 0;
head, tail, size:array[1..n] of int init 0;

append(i, v):
if (size[i] == M)
    throw Exception("Full Queue");
fQueue[tail[i]] = fQueue[tail[i]]+i^(j−1) ∗ v;
tail[i] = (tail[i] + 1)%M;
size[i] = size[i] + 1;

deleteHead(i, v):
if (size[i] == 0)
    throw Exception("Empty Queue");
fQueue[head[i]] = fQueue[head[i]] − i^(j−1) ∗ v;
head[i] = (head[i] + 1)%M;
size[i] = size[i] − 1;

isEmpty(i):
return (size[i] == 0);
```

Fig. 4. Fused Queue Implementation at $F(j)$

mands to the primary state machines are also issued to the fused-SMs. The pseudo-code for such a design is shown in [18].

We now do overhead analysis for both RSM and the fused-SM approach.

Overhead Under Normal Operation: For replication, we require additional nf state machines, f replicas for each of the primary state machine. Each operation requires a message to the primary state machine and f replicas. For fused-SM approach, we require additional f machines. Each operation still requires $f + 1$ messages, one to the primary state machine and f messages from the primary to fused-SMs. The message to the primary state machine is same as for the RSM approach, however messages to the fused-SMs may contain additional state information so that fused machines can execute the event despite availability only of fused data structures.

Assume that the waiting list can have size at most $O(m)$. The RSM approach requires $O(nfm)$ space to tolerate f faults among n machines. The fused-SM approach requires $O(fm+nf)$ space. The component $O(nf)$ is required because we allow $O(1)$ state information for each of the n state machines at the fused-SMs. In the example, we kept $head[i], tail[i]$ and $size[i]$ for each state machine.

The number of events and messages required to be processed at the fused-SM is n times more than the number of events processed by a replica. Thus, if n is large the fused-SMs may become bottleneck. In these cases, one could easily use a hybrid of replicated and fused-SM approach.

Complexity for Recovery after Failure: The RSM approach has minimal overhead for recovery after failure. As soon as a primary machine is detected to be crashed, the replica with the highest id that survives can take over and start functioning as primary.

The recovery overhead in the fused-SM approach is crucially dependent on the number of actual faults t. Let the state of any primary state machine be $O(m)$. First consider the case when t equals 1. The recovery algorithm will require $O(n)$ messages, one from each of the surviving machines of size $O(m)$. It will take $O(nm)$ time to recover the state of the crashed machine. For $t > 1$ faults, we would be required to solve t linearly independent equations. Equivalently, it can be viewed as multiplying the fusion vector with the inverse of the equation

matrix. Since m is large compared to t, we ignore the one time cost of computing the inverse. Thus, we get the overall cost as $O(m(nt + t^2))$.

2.4 Application to Ricart and Agrawala's Algorithm

The state machine for the resource allocator example was based on a centralized algorithm for mutual exclusion. We now show that the technique is also applicable to distributed algorithms such as Ricart and Agarwala's algorithm[9]. Suppose that there are n primary processes $P(1)..P(n)$ that are coordinating access to a *single* critical section. For the RSM based approach, each P_i would need f backups and will result in nf additional state machines (even if they are run on only f additional servers). Since each state machine requires $O(n)$ space to keep track of pending requests, the total space requirement is $O(fn^2)$. The code for the fused-SM based Ricart and Agrawals's algorithm is presented in [18]. With the fused-SM approach, we use f additional state machines with total of $O(fn)$ space. Any request message is also sent to the fused processes which update the fused data structures on behalf of all the processes in the system. Similarly, *okay* messages are also sent to the fused processes.

The non fault-tolerant algorithm requires $2n$ messages per CS invocation. With the RSM approach, every message needs to be sent to f backup processes resulting in $2n(f + 1)$ messages. The Fused-SM approach requires an additional request message and $n - 1$ okays to be sent to any fused process. Thus, the total message requirement is only $2n + nf$, which results in savings of nf messages.

3 Byzantine Faults

So far we had assumed crash faults. We now discuss Byzantine faults where any state machine may change its state arbitrarily. The RSM approach requires that there be $2f$ backup replicas for each primary state machine. Since there are $2f+1$ values available, even if f of them are faulty, the majority will always be correct. When this approach is applied to n different servers, the RSM approach requires additional $2nf$ replicas. For data coding, it is well known that by appending $2f$ parity check symbols, one can recover from f unknown data errors [6]. Can the same ideas be applied to fault-tolerance of state machines?

The additional constraint we have for tolerating Byzantine faults in state machines is that during normal (fault-free) operation, we would like to have as little overhead as possible. Specifically, we would like to avoid the overhead of decoding the state during normal operations. To achieve this goal, we give an algorithm that combines replication with coding theory. We first consider the case of a single Byzantine fault. Next we generalize the algorithm to tolerate f Byzantine faults but assume that each state machine has $O(1)$ state. Finally, we give the algorithm that tolerates f Byzantine faults and each primary state machine may have $O(m)$ state.

3.1 Tolerating Single Byzantine Fault

We start with the case of detecting and tolerating a single Byzantine fault among n primary state machines. The pure RSM approach requires two replicas for every primary machine resulting in $3n$ state machines in all. The pure Fused-SM approach would require $n+2$ machines in all. However, in the pure Fused-SM approach, even the normal operations may be inefficient. For crash faults, the decoding was required only when there was a failure, a low probability event. For Byzantine faults, a pure Fused-SM approach would require decoding even during normal operations just to detect if one of the primary machines is faulty. We now show a hybrid approach that is efficient during normal operation and still requires less number of processes than the RSM approach.

Our algorithm is based on two observations. First, if we have two copies of a primary state machine $P(i)$, then one of these copies is guaranteed to be correct. The RSM approach relies on keeping an additional copy so that majority can be used to determine which is correct. In our approach, we use the concept of *liar detection*. We use the fused-SMs to determine which of the two copies is faulty. The liar detection approach is more efficient in terms of the total number of copies required. The second observation we use is that if two copies of $P(i)$ agree on some value, then that value is guaranteed to be correct (because, there can be at most one Byzantine fault).

Theorem 2. *Let there be n primary state machines, each with $O(m)$ data structures. There exists an algorithm with additional $n+1$ state machines that can tolerate a single Byzantine fault and has the same overhead as the RSM approach during normal operation and additional $O(m+n)$ overhead during recovery.*

Proof. We keep one replica $Q(i)$ for every primary state machine $P(i)$ and a fused-SM $F(1)$ for the entire system. Thus, we keep $2n+1$ state machines in all. During normal operation (when there is no fault), the value of any output at $P(i)$ and $Q(i)$ must be identical. In this case, we do not decode the value from $F(1)$. As soon as $P(i)$ and $Q(i)$ differ for any i, we have detected Byzantine fault in the system. Note that we do not observe the state of $P(i)$ and $Q(i)$ at all events. We only look at the response of $P(i)$ and $Q(i)$ for input events and take action when the response (output) at $P(i)$ differs from $Q(i)$. At this point, we know that either $P(i)$ is correct or $Q(i)$ is correct, but do not know the identity of the liar yet. We now invoke the liar detection algorithm as follows. Given the state of $P(i)$ and $Q(i)$, in $O(m)$ time we can locate the first data of size $O(1)$ that is different in them. We use the fused process $F(1)$ to determine which of these values is correct. This step will require messages of size $O(1)$ from other $n-1$ primary processes. It also requires that the system ensures that all operations that have been performed on the primary state machines have been applied to $F(1)$. Now, in $O(n)$ time the correct value of the data can be determined; therefore, we have the identity and the state of the correct process. The liar process can be killed and a new copy of the correct process can be started.

Observe that in the above algorithm we never decode the data structure at the fused-SM. During normal operations, we only do the encoding. Whenever there is Byzantine fault detected, we use $F(1)$ only to determine which of the copies is correct. We can encode $O(1)$ crucial information to determine whether $P(i)$ or $Q(i)$ is a liar. Also observe that if the fault occurs in the fused machine, it does not affect the overall operation of the system and it is not even detected. If early detection of fault in the fused machine is important for some application, then periodically (or during off-peak period) one could simply reset and recompute the fused process data. Thus, decoding of the fused-SM is not required.

3.2 Tolerating f Byzantine faults in State Machines with $O(1)$ State

To generalize the above algorithm for f faults, we maintain the invariant that there is at least one correct copy in spite of f faults. Therefore, we keep f copies of each of the primary server and f fused copies. Thus, we have total of $n * f + f$ state machines in addition to n primary machines. The only requirement on the fused copies $\{H(j), j = 1..f\}$is that if $H(j)$ is not faulty and if we have $n - 1$ correct values of the primary machines, then the remaining one can be determined using $H(j)$. Thus, a simple xor or sum based fused-SM is sufficient. Even though we are tolerating f faults, the requirement on the fused copy is only for a single fault (because we are also using replication).

The primary copy together with its f replicas are called *unfused* copies. If any of $f + 1$ unfused copies differ, we call the primary server *mismatched*. Let the value of one of the copies be v. The number of unfused copies with value v is called the *multiplicity* of that copy.

We now generalize Theorem 2 for $f \geq 1$ faults. At first, we will assume that the state space of each of the state machines is small. Later, we generalize it to the case when each of the state machine has $O(m)$ state.

Theorem 3. *There exists an algorithm with $fn + f$ backup state machines that can tolerate f Byzantine faults and has the same overhead as the RSM approach during normal operation and additional $O(nf)$ overhead during recovery.*

Proof. We keep f copies for each primary state machine and f overall fused machines. This results in additional $nf + f$ state machines in the system. If there are no faults among unfused copies, all $f + 1$ copies will result in the same output and therefore the system will incur same overhead as the RSM approach.

Our algorithm first checks the number of primary state machines that are mismatched. First consider the case when there is a mismatch between a primary state machine $P(i)$ and its replica for at most one value of $i = 1..n$. Let that primary machine be $P(c)$. Since there are at most f faults, we can conclude that we have the correct state of all other primary state machines $P(i), i \neq c$. Now given the correct state of all other primary machines, we can successively retrieve the state of $P(c)$ from fused machines $H(j), j = 1..f$ till we find one of the unfused machine that has $f + 1$ multiplicity. We have to decode at most f fused machines each at cost of $O(n)$.

Now consider the case when there is a mismatch for at least two primary state machines, say $P(c)$ and $P(d)$. Let $\alpha(c)$ and $\alpha(d)$ be the largest multiplicity among unfused copies of $P(c)$ and $P(d)$ 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 $P(c)$. We now claim that there are at least $f + 1 - \alpha(d)$ liars among unfused copies of $P(d)$ 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 $P(d)$ with multiplicity $\alpha(d)$. If this copy is correct we have $f + 1 - \alpha(d)$ liars. If this value is false, 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 $P(d)$.

By identifying the correct value, we have reduced the number of mismatched primary state machines by 1. By repeating this argument, we get to the case when there is exactly one mismatched primary machine.

Based on the proof of Theorem 3, we get the Algorithm C shown in Figure 5, to tolerate f Byzantine faults with nf replicated and f fused-SMs.

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

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

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

Recovery Algorithm:
Let t be the number of mismatched SMs;
while $t > 1$ do
    choose the copy with largest multiplicity;
    kill all incorrect unfused copies;
    restart them with the chosen copy;
    $t = t - 1$;

// Can assume that t equals one.
// Let $P(c)$ be the mismatched SM
for $(j = 1; j \leq f; j + +)$
    create a new copy using $H(j)$ and $P(i), i \neq c$;
    if (any copy has multiplicity $f + 1$)
        recover to that copy and return;

Fig. 5. Algorithm C: Tolerating f Byzantine faults

In Algorithm C, we had to decode the fused-SMs during the recovery algorithm. The algorithm requires at most f fusion processes to be decoded in the worst case. If there are $t \leq f$ faults, we are guaranteed that after decoding t fused-SMs we will have $f + 1 + t$ unfused copies. At least one of these copies will have multiplicity of $f + 1$ or more. Alternatively, we can try out all the values of

unfused copies of $P(c)$ and $\{P(i), i \neq c\}$ to compute H and thereby determine multiplicity of various copies.

3.3 Tolerating f Byzantine faults for State Machines with $O(m)$ state

We now extend the algorithm to the case when each of the primary state machine has $O(m)$ state. We would like to avoid decoding or encoding the entire fused process. As observed earlier, one of the $f+1$ unfused copies is guaranteed to be correct and it is sufficient to locate this copy using fused copies. We give an algorithm with $O(mf + nt^2)$ time complexity to locate the correct copy. Assume that we are trying to locate the correct copy among unfused copies of $P(c)$.

```
Z:set of copies initially {1..f + 1};
while (all unfused copies in Z not identical)
    w = min{r : ∃p, q ∈ Z : state_p[r] ≠ state_q[r]};
    j = 1;
    while (no copy with multiplicity f + 1)
        create state[w] using H(j) and P(i), i ≠ c;
        j = j + 1;
    endwhile;
    delete other copies from Z;
endwhile;
return any copy from Z;
```

Fig. 6. Locating a Correct Unfused Copy for mismatched $P(c)$: *locate(int c)*

In the algorithm shown in Fig. 6, 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 $1..f+1$ 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 index in which states of copies p and q differ [1]. Either copy p or the copy q (or both) are liars. We now use the fused machines to recreate copies of $state[w]$. Since we have the correct copies of all other primary machines $P(i), i \neq c$, we can use them with the fused copies $H(j), j = 1..f$. Note that the fused copies 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

[1] For simplicity, we view the state of machines as an $O(m)$ array (though in practice it could be any structure with size $O(m)$).

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 $2f+1$ copies. Further, since copies p and q differ in $state[w]$, we are guaranteed to delete at least one of them in each iteration of while. Eventually, the set Z would either be singleton or will contain only identical copies. In either case, the *while* loop terminates and we have located a correct copy.

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 index in each iteration of the outer *while* loop and there are at most t faulty processes giving us the bound of t for the number of iterations of the while loop. In each iteration, creating $state[w]$ requires at most $O(1)$ state to be decoded for each fusion process at the cost of $O(n)$. The maximum number of fused processes that would be required is t. Thus, $O(nt)$ work is required for a single iteration before a copy is deleted from Z. To determine w in incremental fashion requires $O(mf)$ work cumulative over all iterations. Combining these costs we get the complexity of the algorithm to be $O(mf + nt^2)$.

By using the method *locate*, in the recovery algorithm we get the following result – the main result of the paper.

Theorem 4. *Let there be n primary state machines, each with $O(m)$ data structures. There exists an algorithm with additional $nf + f$ state machines that can tolerate f Byzantine faults and has the same overhead as the RSM approach during the normal operation and additional $O(mf + nt^2)$ overhead during recovery where t is the actual number of faults that occurred in the system.*

Theorem 4 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 machine 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 copies it maintains to guarantee that there is at least one correct unfused copy and that faults of any f machines 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.

4 Conclusions

We have presented efficient distributed algorithms to tolerate crash and Byzantine faults of state machines in distributed systems. Our algorithms use a combination of replication and coding theory to achieve efficiency in detection and correction of faults. Our algorithms use fewer backup state machines (and therefore smaller space, and fewer messages in many cases) while providing the same level of fault-tolerance.

Acknowledgements I am thankful to Bharath Balasubramanian, Vinit Ogale and Yogish Sabharwal for discussions on the topic.

References

1. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7) (1978) 558–565
2. Lamport, L.: Using time instead of timeout for fault-tolerant distributed systems. ACM Trans. Program. Lang. Syst. **6**(2) (1984) 254–280
3. Schneider, F.B.: Implementing fault-tolerant services using the state machine approach: A tutorial. ACM Comput. Surv. **22**(4) (1990) 299–319
4. Sivasubramanian, S., Szymaniak, M., Pierre, G., van Steen, M.: Replication for web hosting systems. ACM Comput. Surv. **36**(3) (2004) 291–334
5. MacWilliams, F.J., Sloane, N.J.A.: The Theory of Error-Correcting Codes. North-Holland Publishing Company (1981)
6. van Lint, J.H.: Introduction to Coding Theory. Springer-Verlag (1998)
7. Pease, M., Shostak, R., Lamport, L.: Reaching agreements in the presence of faults. Journal of the ACM **27**(2) (1980) 228–234
8. Raynal, M., Schiper, A., Toueg, S.: The causal ordering abstraction and a simple way to implement it. Information Processing Letters **39**(6) (1991) 343–350
9. Ricart, G., Agrawala, A.K.: An optimal algorithm for mutual exclusion in computer networks. Communications of the ACM **24** (1981)
10. Garg, V.K., Ogale, V.A.: Fusible data structures for fault-tolerance. In: ICDCS, IEEE Computer Society (2007) 20
11. Balasubramanian, B., Garg, V.K.: A fusion-based approach for handling multiple faults in data structures. Technical Report ECE-PDS-2009-001, Parallel and Distributed Systems Laboratory, ECE Dept. University of Texas at Austin (2009)
12. Ogale, V.A., Balasubramanian, B., Garg, V.K.: A fusion-based approach for tolerating faults in finite state machines. In: IPDPS, IEEE (2009) 1–11
13. Patterson, D.A., Gibson, G., Katz, R.H.: A case for redundant arrays of inexpensive disks (raid). In: SIGMOD '88: Proceedings of the 1988 ACM SIGMOD international conference on Management of data, New York, NY, USA, ACM Press (1988) 109–116
14. Chen, P.M., Lee, E.K., Gibson, G.A., Katz, R.H., Patterson, D.A.: Raid: high-performance, reliable secondary storage. ACM Comput. Surv. **26**(2) (1994) 145–185
15. Plank, J.S.: A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. Software – Practice & Experience **27**(9) (1997) 995–1012
16. Luby, M.G., Mitzenmacher, M., Shokrollahi, M.A., Spielman, D.A., Stemann, V.: Practical loss-resilient codes. In: STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing, New York, NY, USA, ACM Press (1997) 150–159
17. Byers, J.W., Luby, M., Mitzenmacher, M., Rege, A.: A digital fountain approach to reliable distribution of bulk data. SIGCOMM Comput. Commun. Rev. **28**(4) (1998) 56–67
18. Garg, V.K.: Implementing fault-tolerant services using fused state machines. Technical Report ECE-PDS-2010-001, Parallel and Distributed Systems Laboratory, ECE Dept. University of Texas at Austin (2010)
19. Plank, J.S., 0002, Y.D.: Note: Correction to the 1997 tutorial on reed-solomon coding. Softw., Pract. Exper. **35**(2) (2005) 189–194
20. Birman, K.P., Joseph, T.A.: Reliable communication in the presence of failures. ACM Transactions on Computer Systems **5**(1) (1987) 47–76