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
garg@ece.utexas.edu

***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 power and storage 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 k different state machines, pure replication based schemes require $k(f + 1)$ replicas to tolerate f crash faults in a system and $k(2f + 1)$ 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 k machines. For Byzantine faults, we propose an algorithm that requires only $kf + f$ additional state machines, as opposed to $2kf$ 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.**

I. 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 [Lam78], [Lam84a] and further elaborated by Schneider in [Sch90] are considered the standard solutions 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 community replication continues to be the dominant approach for fault-tolerance [TDOK04], [SSPvS04]. In this paper, we give an alternate method for fault-tolerance that combines ideas from replication with coding theory [MS81], [vL98] 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 power savings.

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. As an analogy from the coding theory, consider the problem of fault tolerance when a message is being communicated. Our assumption is akin to requiring that the message should have multiple symbols to reap the benefits of the techniques (such as Reed-Solomon).

Pursuing the analogy with the coding theory further, assume that k symbols drawn from a fixed alphabet need to be communicated over a noisy channel. Suppose that we are interested in tolerating f erasures (i.e. detectable corruption of the symbols). It is easy to see that by sending every symbol $f+1$ times, any f erasures can be tolerated. However, the *rate* of such a scheme is quite low and it is well known that instead of sending additional kf symbols, it is necessary and sufficient to send f symbols to tolerate f erasures under a reasonable set of assumptions (for example, by using Reed-Solomon coding). Replication is simple to implement but grossly wasteful.

In this paper, we show how services in a distributed system can be made fault-tolerant using fusion. Given k *different* state machines 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 [PSL80], 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 $n = k + f$ machines can tolerate crash of any f machines in the system. We illustrate our technique on the resource allocation service from [Sch90]. 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 to achieve low overhead during normal operations while reducing the overall storage, power and computation requirement. In particular, we give an algorithm that keeps the overhead of the replicated state machine approach during normal operations but requires only $kf + f$ additional state machines (as opposed to $2kf$ 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.

It is important to note that there are at least two advantages of running fewer backup processes. First, the amount of state and therefore the storage maintained at back up processes is reduced. Second, by reducing the number of processes running in the system we also save on the power required to run the servers that execute the backup processes.

In our earlier work, we have given algorithms for fusible data structures. In particular, [GO07] gives algorithms for arrays, stacks, queues, linked lists etc. This work has been generalized to tolerate multiple faults in [BG09]. 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. In [OBG09], 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, we give techniques that 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 [PGK88], [CLG$^+$94], [Pla97]. As another example, network coding [LMS$^+$97], [BLMR98] 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. By using coding theory techniques [vL98] one can get much better space utilization than, for example, simple replication. These techniques are oblivious to the structure of the data. The details of actual operations on the data are ignored and the coding techniques simply recompute the encoded backups 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, many 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, back up machines can be designed so that they provide fault-tolerance in an efficient manner.

II. Fusible State Machines

We consider the usual model of a distributed system. In this paper, we will be concerned only with process faults. There are k *deterministic* primary state machines, $P(i)$, where i ranges from 1 to k. Throughout the paper, we will use i as an index for the primary state machines with fixed range $1..k$. 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 (RSM) 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 and the fusion approach.

Throughout this paper we assume that channels are reliable and FIFO. We also assume that the messaging system is synchronous, i.e., there is a fixed upper bound for all message arrivals. This assumption makes the crash detection of processes quite simple.

A. Event Counter

To concretize our discussion, we start with k simple state machines, $P(i)$'s, shown in Fig. 1. Each of these k 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 similar $exit(v)$ event.

```
$P(i) :: i = 1..k$
    int $count_i = 0$;

     On event $entry(v)$:
        **if** $(v == i)$ $count_i = count_i + 1$;
     On event $exit(v)$:
        **if** $(v == i)$ $count_i = count_i - 1$;
```

Fig. 1. Event Counter State Machines

First consider the case when f equals one, i.e., we are required to tolerate at most one fault. Since there are k different machines, each with possibly different count, the RSM approach will require k additional machines $\{Q(i), i = 1..k\}$. The machine $Q(i)$ is identical to $P(i)$ and if any of the machine $P(i)$ crashes we can recover it using $Q(i)$.

The Fused-SM based approach for this problem requires a single machine that can recover fault of any of $P(i)$. The machine $F(1)$ is shown in Fig. 2. Instead of tracking all the count variables as in the RSM approach, the fused state machine $F(1)$ tracks the sum of all counts. It increments the variable $fCount_1$ on $entry(i)$ for any i and decrements it for

```
$F(1)$::
    int $fCount_1 = 0$;

    On event $entry(i),\ i = 1..k$:
        $fCount_1 = fCount_1 + 1$;
    On event $exit(i)\ i = 1..k$:
        $fCount_1 = fCount_1 - 1$;
```

Fig. 2. Fusion of Counter State Machines

any $exit(i)$. The recovery procedure is more complex than for replication. It crucially depends on the fact that if any of $P(i)$ crashes, the rest of the machines are still available. If $P(c)$ has failed, then its state $count_c$ can be recovered as

$$count_c = fCount_1 - \sum_{i \neq c} count_i$$

The reader may object to the above method on many grounds. First, the example works for one fault. Can it work for multiple faults? Keeping two copies of $F(1)$ does not tolerate two faults. Second, the state space for each of the machines is infinite. The state-space of the backup machine is also infinite. Are there any real savings? What if the counters are represented using a fixed number of bits? Third, the example is simplistic. Can we do this for any general state machine? We address each of these objections next.

We now show the scheme for tolerating f concurrent faults. For RSM approach, we would have to add kf additional machines. For fusion, we add just f additional machines, $F(1)..F(f)$ as shown in Fig. 3. The code for $F(j)$ is shown in Figure 3. Instead of incrementing the count by one, $F(j)$ increments its count by i^{j-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. 3. Fused State Machine

It can be verified that $fCount_j$ satisfies the following invariant:

$$fCount_j = \sum_i i^{j-1} * count_i \quad for\ all\ j = 1..f$$

We can easily recover states of any f failed state machines using $fCount_j, j = 1..f$. 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 the remaining counts we can get the value of $count_c + count_d$. Using $fCount_2$, we can also 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 $(count_1, count_2, , count_k)$ is viewed as data, $(count_1,count_2,..count_k,fCount_1, fCount_2,..fCount_f)$ 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[MS81]. The unique solvability of all the counts is easy to show.

Theorem 1: Suppose $\boldsymbol{x} = (count_1, count_2, , count_k)$ 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 k values out of $\boldsymbol{y} = (count_1, count_2, ..count_k, fCount_1, fCount_2, ..fCount_f)$ the remaining values in $\boldsymbol{x}$ can be uniquely determined.

Proof: View the state of the primary state machines as $\boldsymbol{x}$, a $1 \times k$ vector. The vector $\boldsymbol{y}$ is a $(k+f) \times 1$ vector consisting of the current state of the primary machines adjoined with the states of the fused machines. By definition of $fCount$, we can view the system as $\boldsymbol{y} = \boldsymbol{xG}$ where G is $k \times (k + f)$ matrix which is $k \times k$ identity matrix adjoined with the transpose of Vandermonde matrix. Let f values of $\boldsymbol{y}$ be erased. By supressing the indices corresponding to the lost values in $\boldsymbol{y}$, we get the vector $\boldsymbol{y'}$ of size k. By deleting corresponding columns in $\boldsymbol{G}$, we get a square matrix $\boldsymbol{M}$ of size $k \times k$. We know that $y' = xM$. Furthermore, when elements of G are from the (infinite) field of real numbers, it is known that M is a nonsingular matrix for all choices of the columns in G[MS81]. Hence M is invertible, and x can be obtained as $y'M^{-1}$. ■

So far we had assumed that by adding numbers we do not get overflow. However, in real computers the count would be stored using a fixed number of bits. While the counts in the primary processes may be represented using this fixed size, the value of $fCount$'s may overflow. There are two possible approaches to tackle this problem. The first possibility is to do all the arithmetic, i.e. addition (subtraction), and multiplication (division) in finite Galois field as typically done in coding theory [MS81]. 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 k rows form an identity matrix [P05]. Note that finding the inverses of the special matrices, such as Cauchy or Vandermonde, does not require the cubic time complexity and can be achieved in quadratic time complexity.

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. Alternatively, in many applications, the semantics of the variable may dictate that we do not get overflow. For example, suppose that we have $2^3 = 8$ primary processes and we want to tolerate two crashes. Suppose we also know that counts never exceeds 2^{20}. Then, the value of $fCount_2$ is at most $\sum_i i*2^{20} \leq 8*8*2^{20}$ which requires only 26 bits. We can then use standard arithmetic instead of Galois arithmetic. Standard arithmetic may be faster than Galois arithmetic in standard current processor architectures.

B. Causal Ordering

Another objection to program $F(1)$ is that it may not generalize to arbitrary events. We show in this section that any set of state machines with similar structure can be fused. The overhead associated with maintaining the fused state machines depend upon the specific application.

We first generalize our primary programs to contain not one variable but a set of data structures. We continue to assume that all primary machines are instances of similar state machines and therefore every state machine has an instance of that variable. A more general state machine is shown in Fig. 4.

```
$P(i) :: i = 1..k$
$var_1, var_2...var_v$: set of variables;

On receiving an event($arg_1, arg_2, ...arg_r$):
    update variables;
    send (UPDATE, $varList$, list of $\delta$) to $F(j)$'s;

$F(j) :: j = 1..f$
$fVar_1, fVar_2...fVar_v$: set of variables;

On receiving UPDATE from $P(i)$;
    for all $var_q$ in $varList$ with change $\delta$
        $fVar_q = fVar_q + i^{j-1} * \delta$
```

Fig. 4. A General Fusible State Machine Approach for Simple Variables

The above method will work when the state machine manipulates simple variables or arrays where the number of variables (or the number of entries in the array) changed due to event is small. For example, suppose that we are running a *causal ordering* algorithm [BJ87] in a group of k processes. Consider the version described by Raynal, Schiper, and Toueg [RST91]. 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. The entry $M[i, r]$ is incremented to reflect the fact that one more message has been sent from $P(i)$ to $P(r)$. Whenever messages are received by the communication system at $P(i)$, they are first checked for eligibility before delivery to $P(i)$. If a message is not eligible it is simply buffered until it becomes eligible. 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. The algorithm for process $P(i)$ and fused process $F(j)$ is given in Fig. 5. It only requires $P(i)$'s to send an "M-Update" message with incremental changes in entries of the matrix.

```
$P(i) :: i = 1..k$
$M$:array[1..$k$, 1..$k$] of int initially $\forall q, r : M[q, r] = 0$;

To send a message to $P(r)$:
    piggyback $M$ as part of the message;
    $M[i, r] := M[i, r] + 1$;
    send(M-Update, $\{(i, r, 1)\}$) to $F(j)$;

To receive a message with matrix $W$ from $P(r)$
    **enabled if** $\forall q : M[q, i] \geq W[q, i]$
    $M := max(M, W)$;
    $M[r, i] := M[r, i] + 1$;
    send(M-Update, list of $\delta$) to $F(j)$;


$F(j) :: j = 1..f$
$M$:array[1..$k$, 1..$k$] of int initially $\forall q, r : M[q, r] = 0$;

On receiving (M-Update, list of $\delta$) from $P(i)$
    for all $(q, r)$ with change $\delta$
        $M[q, r] = M[q, r] + i^{j-1} * \delta$
```

Fig. 5. A fault-tolerant algorithm for causal ordering of messages

Note that the storage requirement for fused processes is $O(fk^2)$ as opposed to $O(fk^3)$ required by a replication based algorithm. A similar algorithm can be used to recover vector clocks[Mat89], [Fid89] of faulty processes in distributed systems. The same approach will also work for Memory State Machine Server in [Sch90].

C. 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 [Sch90]. Assume that there are k different type of resources that can only be used in mutually exclusive fashion. The state machine $P(i)$ shown in Fig. 6 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. The pid of a real user is assumed to be positive and the value of 0 denotes that the resource is currently idle.

Suppose that we want to tolerate one fault in any of these k machines. Whenever, the variable $user$ changes we can

```
$P(i) :: i = 1..k$
    $user$: int initially 0;
    $waiting$: queue of int initially null;

On receiving acquire from client $pid$
    if ($user$ == 0) {
        send(OK) to client $pid$; $user$ = $pid$;}
    else append($waiting$, $pid$);

On receiving release
    if ($waiting$.isEmpty())  user = 0;
    else { user = $waiting$.head();
        send(OK) to $user$;
        $waiting$.removeHead(); }
```

Fig. 6. Resource Allocator State Machine from [Sch90]

```
$P(i) :: i = 1..k$
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 { append($waiting, pid$);
        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 $F(j)$'s }
    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 ; }
```

Fig. 7. Algorithm A: Primary State Machine

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[GO07]. 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. 7. 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. It uses three types of messages: USER, ADD-WAITING, and DEL-WAITING. The message type USER includes any *incremental* change in the user variable. The message type ADD-WAITING includes the id of the user that has been added to the queue and similarly the DEL-WAITING message includes the user that has been deleted from the queue (and allocated the resource). 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.

The code for the fused state machine is shown in Fig. 8. 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 [GO07]).

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

D. Algorithm B: Alternative Design of Fusion Machines

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 which is closer to the structure of RSM. Suppose that the commands to the primary machines are also issued to the fusion machines. In the alternative design, the primary machines do not send all the incremental changes to the fusion machines. They only send minimal change in state that cannot be determined by the fusion machines themselves. In the resource allocator example, the fused machine will be required to send the update only of the user variable when it is extracted from the waiting list. Since fusion processes keep the lists in fused form, they cannot determine it directly without the help of primary processes. In this design, the primary processes will be same as in Fig. 6 used for RSM approach except that the statement "send(OK) to $waiting.head()$;" will be changed to " send(OK, $user$) to $user$ and fusion processes;"

The fusion process $G(1)$ is shown in Figure 9.

We now do overhead analysis for both RSM and the fused-SM approach. It is important to make a distinction between two cases: the normal case and the worst case. As well-known in system design, it is important to be fast for the normal case whereas it is sufficient to guarantee recovery for the worst case [Lam84b].

Overhead Under Normal Operation: As before, we consider the case when there are k primary state machines and are required to tolerate f faults. For replication, we require additional kf 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 state machines. The message to the primary state machine is same as for the RSM approach, however messages to the fused state machines may contain additional state information so that fused machines can execute the event despite availability

```
$F(j) :: j = 1..f$
    $fuser$: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, $i$, $user$)
    $fwaiting.deleteHead(i, user)$;

    // Fused queue implemented using array ;
    $fQueue$: array[$0..M-1$] of int initially 0;
    $head$, $tail$, $size$: array[$1..k$] of int initially 0;

$append(i, pid)$;
    **if** ($size[i] == M$)
        throw Exception("Full Queue");
    $fQueue[tail[i]]$ = $fQueue[tail[i]]$ + $i^{j-1} * pid$;
    $tail[i]$ = $(tail[i] + 1)\%M$;
    $size[i] = size[i] + 1$;

$deleteHead(i, pid)$;
    **if** ($size[i] == 0$)
        throw Exception("Empty Queue");
    $fQueue[head[i]]$ = $fQueue[head[i]]$ - $i^{j-1} * pid$;
    $head[i] = (head[i] + 1)\%M$;
    $size[i] = size[i] - 1$;

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

Fig. 8. Algorithm A: Fused State Machine for Resource Allocation

```
$G(j) :: j = 1..f$
$gUser$:int initially 0;
$gWaiting$: fused queue; (see Fig. 8)

On receiving acquire resource $i$ from client $pid$
    if ($gWaiting.isEmpty(i)$)
        $gUser = gUser + pid * i^{j-1}$;
    else $gWaiting.append(i, pid)$;

On receiving release resource $i$ from client $pid$
    $gUser = gUser - pid * i^{j-1}$;

On receiving (OK, $pid$) from $P(i)$
    $gWaiting.deleteHead(i, pid)$;
    $gUser = gUser + pid * i^{j-1}$;
```

Fig. 9. Algorithm B: Alternative Design of Fused State Machines

only of fused data structures. In the example of resource allocator, for Algorithm A, the primary machine sends an additional message of size $O(\log k)$ every event (for the release event, our code shows two messages for clarity, but these two messages can easily be combined into a single message). For Algorithm B, the clients send $f+1$ messages as for RSM; however, additional f messages may be required to communicate "crucial" state to the fused state machines.

Let us analyze the space overhead for the RSM approach for the resource allocator example. Assume that the waiting list can have size at most $O(m)$. The RSM approach will require $O(kfm)$ space to tolerate f faults among k machines. The fused-SM approach requires $O(fm + kf)$ space. The component $O(kf)$ is required because we allow $O(1)$ state information for each of the k state machines at the fused state machines. 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 state machine is k times more than the number of events processed by a replica. Thus, if k is large the fused state machines 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. Note that the recovery time is independent of the number of state machines k in the system.

The recovery overhead in the fused-SM approach is crucuially dependent on the number of actual faults t. Let the state of any primary state machine be $O(m)$. First consider the most probable case, i.e. t equals 1. The recovery algorithm will require $O(k)$ messages, one from each of the surviving machines of size $O(m)$. It will take $O(km)$ 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 $O(m(kt + t^2))$.

E. Alternative Methods for Systematic Coding

In this section, we briefly outline some other systematic linear coding algorithms that can be employed in our setting. In Section II-A, we had used transpose of Vandermonde matrix. When the value of the variable $count_i$ changed by δ, we simply added $i^{j-1}\delta$ to the code stored at $F(j)$. Alternatively, we could add $j^{i-1}\delta$ to the code stored at $F(j)$ (i.e., switch the roles of i and j).

It can be verified that $fCount_j$ now satisfies the invariant:

$$fCount_j = \sum_i j^{i-1} * count_i \ for\ all\ j = 1..f$$

Another way to look at $fCount_j$ is to consider the polynomial

$$p(x) = \sum_i count_i * x^{i-1}$$

Then, $fCount_j$ is just the evaluation of the polynomial $p(x)$ on j. It can be verified that this method also allows the system to recover from f faults. However, we have preferred the transpose of the Vandermonde rather than Vandermonde matrix for the following reason. We expect that for practical applications k would be much larger than f. In our original proposal the computation required for smaller number of faults is much simpler. For small values of f, the value of $fCount_j$ would be at most $O(f \log k + m)$ bits where m is the maximum number of bits required to store any $count_i$.

Another possibility is to use Cauchy Matrices as used in [Rab89]. It is well known that any square submatrix of a Cauchy Matrix is also a Cauchy matrix (and therefore non-singular). The advantage of Cauchy Matrices is that they allow systematic coding for finite fields. We can choose the generator matrix G for coding as $[I|C]$ where C is a Cauchy Matrix. Every square submatrix of G is invertible. Vandermonde matrices do not satisfy this property for finite fields. However, variants of these matrices do satisy the required property[LF03]. We have chosen to use transpose of Vandemonde matrices for their simplicity in explaining the concepts in the context for infinite precision fields and the discussion will carry over to variants such as Cauchy Matrices.

III. Byzantine Faults

So far we had focused on 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 k different servers, the RSM approach requires additional $2kf$ replicas. For data coding, it is well known that by appending $2f$ parity check symbols, one can recover from f unknown data errors. 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.

A. Tolerating Single Byzantine Fault

We start with the case of detecting and tolerating a single Byzantine fault among k primary state machines. The pure RSM approach will require two replicas for every primary machine resulting in $3k$ state machines in all. The pure Fused SM approach would require $k + 2$ 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 state machines 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 k primary state machines, each with $O(m)$ data structures. There exists an algorithm with additional $k + 1$ state machines that can tolerate a single Byzantine fault and has the same overhead as the RSM approach during normal operation and additional $O(m + k)$ overhead during recovery.

Proof: We keep one replica $Q(i)$ for every primary state machine $P(i)$ and a fused state machine $F(1)$ for the entire system. Thus, we keep $2k + 1$ 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. 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 determine the data of size $O(1)$ that is different in them and therefore responsible for different outputs. 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 $k - 1$ primary processes. In $O(k)$ time the correct value of the data can be determined. We now have the identity and therefore 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 state machine. 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 state machine is not required.

B. 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 $k * f + f$ state machines in addition to k 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 $k-1$ correct values of the primary machines, then the remaining one can be determined using $H(j)$. Thus, a simple xor or sum based fused state machine 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 $fk+f$ backup state machines that can tolerate f Byzantine faults and has the same overhead as the RSM approach during normal operation and additional $O(kf)$ overhead during recovery.

Proof: We keep f copies for each primary state machine and f overall fused machines. This results in additional $kf+f$ 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 primary state machine $P(i)$ and its replica for at most one value of $i = 1..k$. 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 will have to decode at most f fused machines each at cost of $O(k)$.

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. ■

Remark: In the proof of Theorem 3, whenever there are more than one mismatched primary machines c and d, such that $\alpha(c) \geq \alpha(d)$, it can be shown that $\alpha(c) \geq \lceil f/2 \rceil + 1$. The claim follows because either the unfused copies of $P(c)$ or $P(d)$ has at least $\lceil f/2 \rceil + 1$ correct copies; otherwise we show that the system has more than f faults. There are $2f+2$ unfused copies of $P(c)$ and $P(d)$. Since there are at most f faults; there are at least $f+2$ correct copies. Therefore, either there are $\lceil f/2 \rceil + 1$ correct copies among unfused copies of $P(c)$ or those for $P(d)$. Hence, there is at least one copy with multiplicity at least $\lceil f/2 \rceil + 1$.

Based on the proof of Theorem 3, we get the algorithm C shown in Figure 10, to tolerate f Byzantine faults with kf replicated and f fused state machines.

In Algorithm C, we had to decode the fused state machines 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 state machines 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.

C. 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 $O(mf + kt^2)$ time complexity to locate the correct copy. Assume that we are trying to locate the correct copy among unfused copies of $P(c)$.

In the algorithm shown in Fig. 11, 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

```
*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 any 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 state machines;
**while** $t > 1$ do
    choose the copy with largest multiplicity;
    kill all incorrect unfused copies;
    restart killed processes with the chosen copy;
    $t = t - 1$;

// Can assume that $t$ equals one.
// Let the mismatched machine be $P(c)$
**for** $(j = 1; j \leq f; j++)$
    create 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. 10. **Algorithm C:** Tolerating f Byzantine faults

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 multiplicty 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 $2f+1$ 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.

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

```
*locate(int c)*::
$Z$:set of copies initially $\{1..f+1\}$;

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

Fig. 11. Locating A Correct Unfused Copy

We now analyze the time complexity of the procedure *locate*. Assume that there are $t \leq f$ actual faults that occured. 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 $O(k)$. The maximum number of fused processes that would be required is t. Thus, $O(kt)$ 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 $O(mf + kt^2)$.

By using the method *locate*, in the recovery algorithm we get the following result.

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

IV. 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 processes while providing the same level of fault-tolerance.

V. Acknowledgements

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

References

[BG09] Bharath Balasubramanian and Vijay K. Garg. 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.

[BJ87] K. P. Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, 1987.

[BLMR98] John W. Byers, Michael Luby, Michael Mitzenmacher, and Ashutosh Rege. A digital fountain approach to reliable distribution of bulk data. *SIGCOMM Comput. Commun. Rev.*, 28(4):56–67, 1998.

[CLG$^+$94] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. Raid: high-performance, reliable secondary storage. *ACM Comput. Surv.*, 26(2):145–185, 1994.

[Fid89] C. J. Fidge. Partial orders for parallel debugging. *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, 24(1):183–194, January 1989.

[GO07] Vijay K. Garg and Vinit A. Ogale. Fusible data structures for fault-tolerance. In *ICDCS*, page 20. IEEE Computer Society, 2007.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[Lam84a] Leslie Lamport. Using time instead of timeout for fault-tolerant distributed systems. *ACM Trans. Program. Lang. Syst.*, 6(2):254–280, 1984.

[Lam84b] Butler W. Lampson. Hints for computer system design. *IEEE Software*, 1(1):11–28, 1984.

[LF03] Jérôme Lacan and Jérôme Fimes. A construction of matrices with no singular square submatrices. In Gary L. Mullen, Alain Poli, and Henning Stichtenoth, editors, *International Conference on Finite Fields and Applications*, volume 2948 of *Lecture Notes in Computer Science*, pages 145–147. Springer, 2003.

[LMS$^+$97] Michael G. Luby, Michael Mitzenmacher, M. Amin Shokrollahi, Daniel A. Spielman, and Volker Stemann. Practical loss-resilient codes. In *STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 150–159, New York, NY, USA, 1997. ACM Press.

[Mat89] F. Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms: Proc. of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B.V. (North-Holland), 1989.

[MS81] F J MacWilliams and N J A Sloane. *The Theory of Error-Correcting Codes*. 1981.

[OBG09] Vinit A. Ogale, Bharath Balasubramanian, and Vijay K. Garg. A fusion-based approach for tolerating faults in finite state machines. In *IPDPS*, pages 1–11. IEEE, 2009.

[P05] James S. Plank and Ying Ding 0002. Note: Correction to the 1997 tutorial on reed-solomon coding. *Softw., Pract. Exper.*, 35(2):189–194, 2005.

[PGK88] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (raid). In *SIGMOD '88: Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, pages 109–116, New York, NY, USA, 1988. ACM Press.

[Pla97] J. S. Plank. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. *Software – Practice & Experience*, 27(9):995–1012, September 1997.

[PSL80] M. Pease, R. Shostak, and L. Lamport. Reaching agreements in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.

[Rab89] Michael O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *J. ACM*, 36(2):335–348, 1989.

[RST91] M. Raynal, A. Schiper, and S. Toueg. The causal ordering abstraction and a simple way to implement it. *Information Processing Letters*, 39(6):343–350, July 1991.

[Sch90] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[SSPvS04] Swaminathan Sivasubramanian, Michal Szymaniak, Guillaume Pierre, and Maarten van Steen. Replication for web hosting systems. *ACM Comput. Surv.*, 36(3):291–334, 2004.

[TDOK04] Fathi Tenzakhti, Khaled Day, and M. Ould-Khaoua. Replication algorithms for the world-wide web. *J. Syst. Archit.*, 50(10):591–605, 2004.

[vL98] J. H. van Lint. *Introduction to Coding Theory*. Springer-Verlag, 1998.