ISSN (Online) : 2319 - 8753
ISSN (Print) : 2347 - 6710

International Journal of Innovative Research in Science, Engineering and Technology

*Volume 3, Special Issue 3, March 2014*

**2014 International Conference on Innovations in Engineering and Technology (ICIET'14)**

**On 21st& 22nd March Organized by**

**K.L.N. College of Engineering, Madurai, Tamil Nadu, India**

# Fused Data Structure for Tolerating Faults in Distributed System

Jeva Kumar M, Golda Jeyasheeli P

PG Scholar, Dept of Computer Science Engineering, Mepco Schlenk, Engineering College, Sivakasi, India.

Assistant Professor, Dept of Computer Science Engineering, Mepco Schlenk, Engineering College, Sivakasi, India.

**Abstract**-In Distributed systems, servers are prone to crash faults in which the data structures (queue, stack, etc) may crash, leading to a total loss in state. Hence it is necessary to tolerate crash faults in distributed system.Replication is the prevalent solution to tolerate crash faults .In replication, entire copy of the original data is taken and stored. Every update to original data reflects changes in the replicated data. Replication is used to ensure consistency, improve reliability, fault-tolerance and accessibility.To tolerate f crash faults among n distinct data structures, replication requires f replicas of each data structure, resulting in nf additional backups.Maintaining nf additional backups for n distinct data structures requires more space. In this project, fused data structure is used for backups which can tolerate f crash faults using just f additional fused backups.

**Keywords:** Distributed systems, fault tolerance, data structures.

## I.INTRODUCTION

In distributed systems, servers maintain large instances of data structures such as linked lists, queues, and hash tables for handling list of pending request from theclients. These servers are prone to crash faults, leading to a total loss in state. Active replication [6],[7] is the mostly preferred solution. To tolerate f crash faults among n given data structures, replication maintains f + 1 replicas of each data structure, resulting in a total of nf backups. For example, consider a set of lock servers that maintain and coordinate the use of locks. Such a server maintains a list of pending requests in the form of a queue. To tolerate four crash faults among, say six independent lock servers each hosting a queue, replication requires four replicas of each queue, resulting in a total of 24 backup queues. For large values of n, this is expensive in terms of the space required by the backups as well as power and other resources to maintain the backup processes.Space efficient can be achieved by using Coding theory [4],[9].

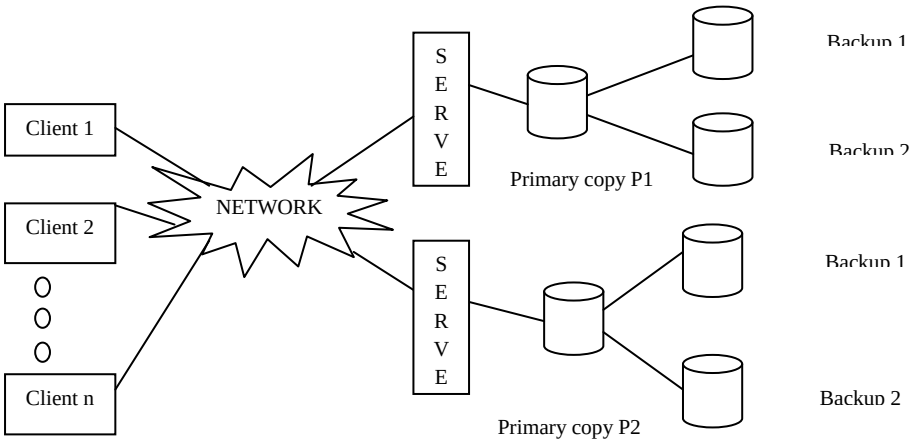


Fig. 1.1: Distributed System Architecture

Fusion is a technique which achieves space efficiency by tolerating f crash faults with f fused backups [2],[3]. In fusion, the backup copies are not identical to the given data. Fused stack is maintained at the backups. The data in the fused stack are in the form of fused state performed by an operation either addition or XOR. As a result space efficient is achieved and also crash faults can be tolerated with minimum number of backups than replication. Hence fused backups are space-efficient, at the mean while they cause very little overhead for normal operation [1],[7],[8].

## II. SYSTEM DESIGN

The system consists of independent servers hosting data structures. Let n denote given data structures, alsoreferred as primaries $P_1$ ...$P_n$. The backup data structures that are generated based on the idea of fusing the primary data are referred as fused backups or fused data structures and it is denoted as $F_1$ ...$F_t$. The operator used to combine primary data is called the fusion operator. The operator may be XOR or simple sum. The number of fused backups, t, depends on the fusion operator and the number of faults that need to be tolerated. Fused backup for binary search tree is shown in Fig.2.1. Here $P_1$ and $P_2$ are the primaries, $F_1$ and $F_2$ are the fused backups and the fusion operator is sum. Here binary search tree is used as primary data structure. Auxiliary nodes are maintained in list.

## III. IMPLEMENTATION

### *A. Insertion*

When key-value pair is sent to primary $P_i$for which the key is not already present in $P_i$, then a new node containing this key-value pair is created and inserted it into the primary binary search tree $P_i$. A pointer to this node is added at the end of the aux list (auxList). The primary $P_i$ sends the key-value pair and the old value (if key already exist) to all fused backups. Each fused backup maintains a stack (data Stack) that contains the primary elements in the coded form. On receiving values from $P_i$, if the key is not already present, the backup insert the value at tos[i]. Else if key already present then backup erase the old value and insert the new value. To maintain order information, the backup inserts a pointer to the newly updated fused node, which points to the corresponding key in the index structure (indexList[i]). Fig. 3.1 shows the state of $P_1$ and $F_1$ after the insert of (3, $a_1$*).

At $F_1$, the value of the third node is updated to ($a_1$* + $b_3$) and a pointer to this node is inserted at indexList[1]. The identical operation is performed at $F_2$, with the only difference is that the third fused node is updated to ($a_1$* - $b_3$). It is to be noted that the aux list at $P_1$ specifies the exact order of elements maintained at the backup stack ($a_1$ -> $a^2$ -> $a_1$*) and indexList[1] at the fused backup specifies the order of elements maintained at $P_1$ ($a_1$ -> $a_1$* -> $a_2$).

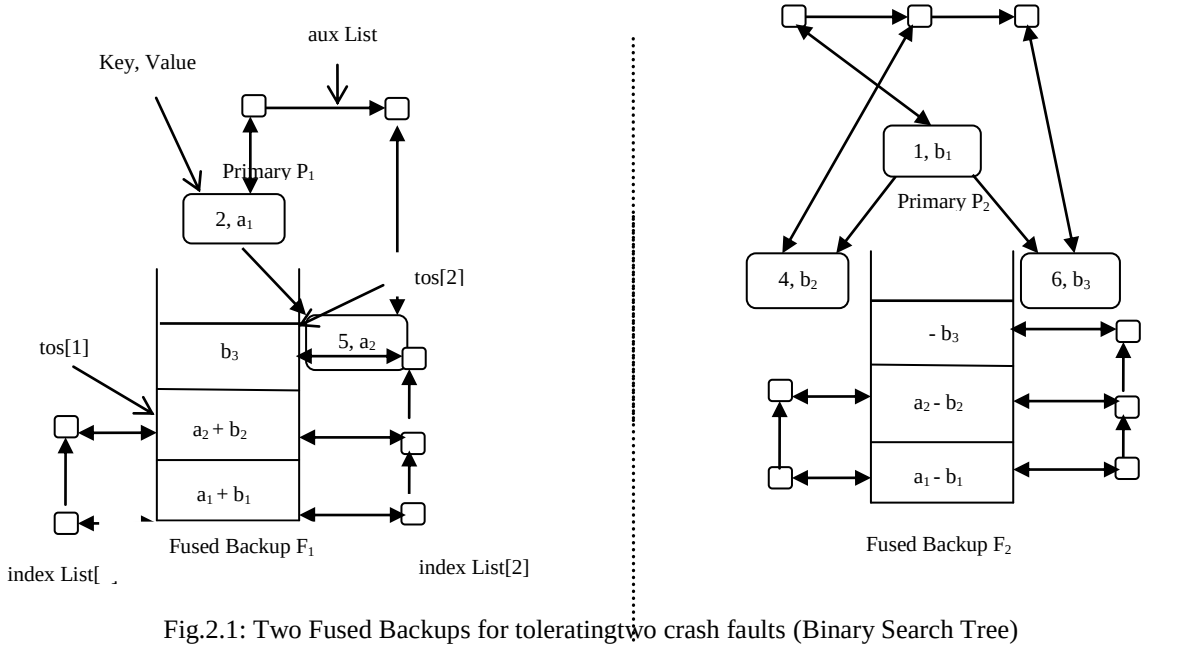


Fig.2.1: Two Fused Backups for toleratingtwo crash faults (Binary Search Tree)

### *B. Deletion*

When a key-value pair is deleted from the primary, corresponding node in the auxiliary list that contains a pointer to this key-value pair is deleted and final auxiliary node is shifted to this position. Hence, the primary knows exactly which value to send with every delete.The node associated with the given key is deleted from the primary and its value that is need to be sent to all fused backups is obtained. Along with this obtained key-value pair, the value pointed by the tail node of the aux list is also sent to the fused backups. This corresponds to the top-most element of $P_i$ at the backup stack and hence helpful for the shift operation that will be performed at the backup. After sending these values, the primary shifts the final node of the aux list to the position of the aux node pointing to the deleted key-value pair, to mimic the shift of the final element at the backup.

At the backup, using the key sent by the primary $P_i$, corresponding fused node is obtained that contains the element of $P_i$ associated with the key. The value of the

node is updated with the top-most element (sent by the primary) to simulate the shift. The pointers of indexList[i] are updated to reflect this shift. Fig. 3.2 shows the state of $P_1$ and $F_1$ after the delete of $b_1$. If the size of the primary or fused nodes is in the order of megabytes, the size of the index structures or auxiliary structures is just in the order of bytes (next pointers). So the space overhead of maintaining these auxiliary/index structures is negligible.

*C. Correcting Crash Faults*

To correct crash faults, the client needs to acquire the state of all the available data structures, both primaries and backups. When crash fault occurs, it is impossible to retrieve data from the crashed data structure. As a result there is a need to retrieve it from fused backup. Since fused node contains data in fused state, there is a need to decode the fused node in order to obtain the original data. The decoding method depends upon the erasure code used.

Let us assume two crash fault had occurred, at both primaries $P_1$ and $P_2$. In order to retrieve the contents of both primaries following operation is performed.

Obtain values from both fused backups $F_1$ and $F_2$. i.e $(a_i + b_j)$ and $(a_i - b_j)$

To retrieve values of P1 perform $[(a_i + b_j) + (a_i - b_j)] / 2$

To retrieve values of P2 perform $[(a_i + b_j) - (a_i + b_j)]/2$

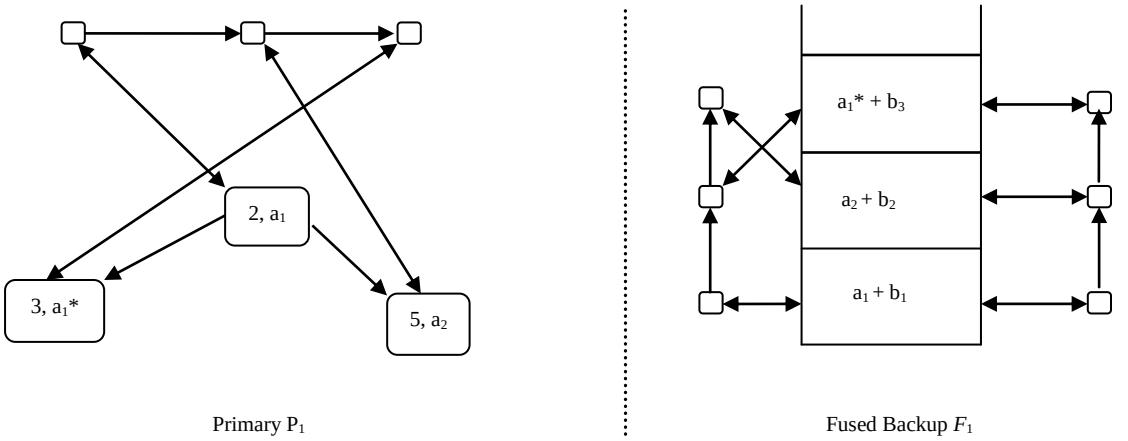


Fig. 3.1: INSERTION (Binary Search Tree)

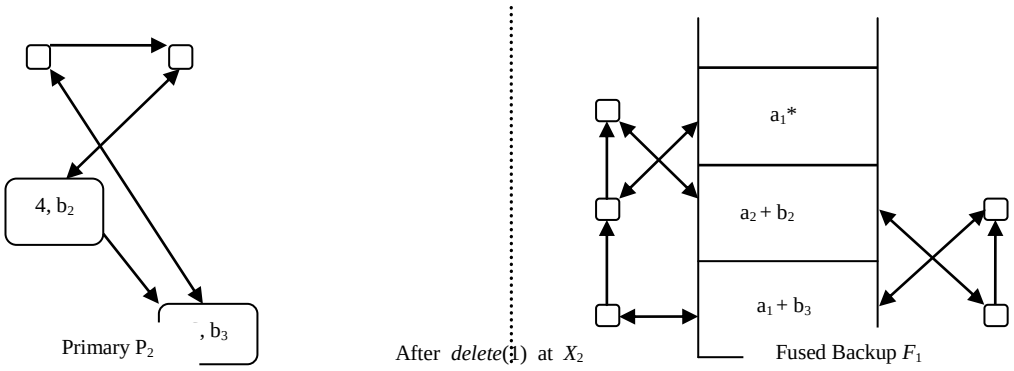


Fig. 3.2: DELETION (Binary Search Tree)

## IV. PERFORMANCE EVALUATION

In this section, the main differences between replication and fusion were summarized. Throughout this section, assume n primary data structures, containing at most O(m) nodes of size O(s) each. Each primary can be updated in O(x) time. Assume that the system can correct f crash faults and t is the actual number of crash faults that occured. Table 4.1 shows performance analysis of both replication and fusion.

TABLE 4.1: Performance Evaluation

| | Replication | Fusion |
|---|---|---|
| Number of primary copies 'n' | 2 | 2 |
| Number of crash fault tolerance 'f' | 2 | 2 |
| **Number of Backups required** | **4** | **2** |
| Size in terms of nodes 'm' (each primaries) | 100 | 100 |
| Size of data 's' in each node | 20 bytes | 20 bytes |
| **Backup Space** | **nmsf=8000 bytes** | **msf=4000 bytes** |
| Number crashes consider as occurred 't' | 2 | 2 |
| Recovery Time | O(mst)=O(4000) | O(ms(t^2)n)=O(16000) |
| Normal (fault free) Operation Messages | 2 msgs, 20 bytes each | 2 msgs, 40 bytes each |
| Recovery Messages 'ms' size each | t msgs (2 msg, 2000 bytes each) | n+f-tmsgs (2 msg, 2000 bytes each) |

*a. Number Of Backups*

To correct f crash faults among n primaries, fusion requires f backup data structures as compared to the nf backup data structures required by replication. Let number of primary data structure be 10. Then the graph (Fig.4.1) compares replication and fusion in terms of number of backups required.

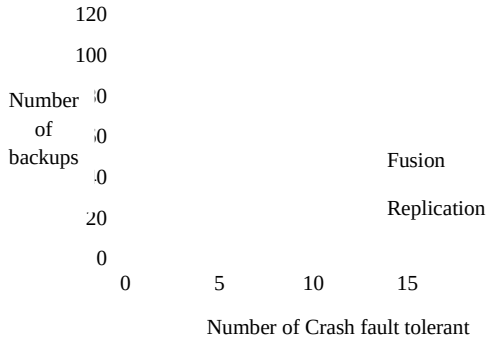


Fig.4.1: Number of backups

*b. Backup Space*

For crash faults, the total space occupied by the fused backups is msf (f backups of size ms each) as compared to nmsf for replication (nf backups of size ms each). Let m = 100 nodes, size of data s=20 bytes and n=10 distinct primary data structure. Then following graph (Fig.4.2) compare both replication and fusion in terms of backup space.

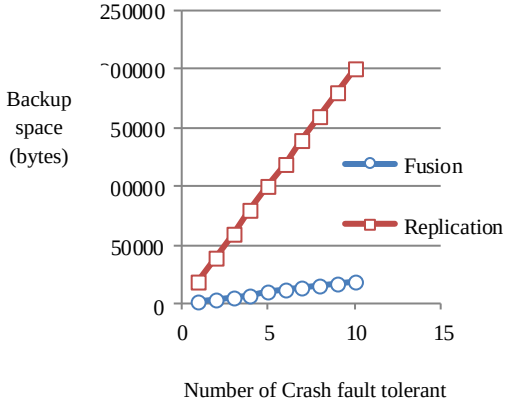


Fig.4.2: Backup Space

*c. Recovery Messages*

This refers to the number of messages that need to be exchanged once a fault has been detected. When t crash faults are detected, in fusion, n + f - t messages of size O(ms) each are exchanged. In replication only t messages of size O(ms) each is required. Let n=10 and f=10, following graph (Fig. 4.3) compares replication and fusion in terms of number of recovery messages needed.

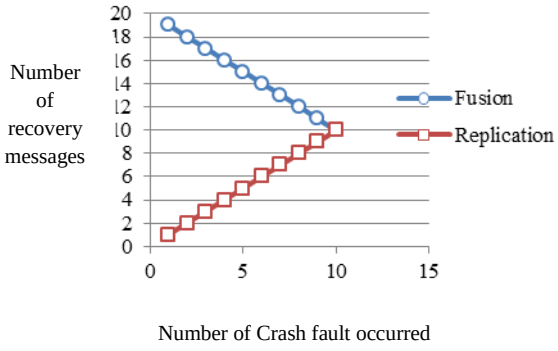


Fig. 4.3: Number of recovery messages

*d. Fault Free Operation*

The fused backups in this system can be updated with the same time complexity as that for updating the corresponding primary, i.e., O(x).

*e. Fault Free Operation Messages*

The number of messages that the primary needs to send to the backups for any update is known as fault free operation messages. In fusion, for crash faults, every update sent to the primary needs to be sent to f backups. The size of each message is 2s since there is a need to send the new value and old value to the backups. For deletes, the size of each message is 2s since there is a need to send the old value and the value of the top-of-stack element. Hence, for crash faults, in fusion, for any update, f messages of size 2s need to be exchanged. For replication, in inserts, only the new value needs to be sent to the f copies of the primary and for deletes, only the key to be deleted needs to be sent. Hence, for crash faults in replication, for any update f messages of size at most s need to be exchanged.

*f. Recovery Time*

Recovery time is measured as the time taken to recover the state of the crashed data structures after the client obtains the state of the relevant data structures. In the case of fusion, to recover from t crash faults, there is a need to decode the backups with total time complexity $O(mst^2n)$. For replication, this is only O(mst). Thus, replication perform much better than fusion in terms of the time taken for recovery.

*g. Update Time*

Fusion has more update overhead as compared to replication. In fusion, the update time at a backup includes the time taken to locate the node to update plus the time taken to update the node's code value. The code update time was low and almost all the update time was spent in locating the node. Hence, optimizing the update algorithm can reduce the total update time.

## V. CONCLUSION

Given n primaries, a fusion-based technique is presented for crash fault tolerance that guarantees O(n) savings in space as compared to replication with almost no overhead during fault free operation. Generic design of fused backups and their implementation for the data structures in the c++ that includes Linked List and Binary Search tree were provided. The main feature of this work is compared with replication. The performance result evaluation confirms that fusion is extremely space efficient while replication is efficient in terms of recovery and the size of the messages that need to be sent to the backups. Using the concepts presented in this project, there is a possibility for an alternate design using a combination of replication and fusion-based techniques. Thus here by conclude that fusion achieves significant savings in space, power, and other resources.

## VI. FUTURE WORK

This work can be extended to more number of data structure other than linked list and binary search tree.Update time in fused backup can be reduced by optimizing the update algorithm.Recovery time can be reduced by using efficient recovery algorithm.

## REFERENCES

[1]BharathBalasubramanian and Vijay K. GargG, "Fault Tolerance in Distributed Systems Using Fused Data Structures," IEEE Transactions On Parallel And Distributed Systems, Vol. 24, No. 4, pg no. 701-715, April 2013.

[2]Balasubramanian B and V.K. Garg, "Fused Data Structure Library (Implemented in Java 1.6)," Parallel and Distributed Systems Laboratory, http://maple.ece.utexas.edu, 2010.

[3]Balasubramanian B and V.K. Garg, "Fused Data Structures for Handling Multiple Faults in Distributed Systems," Proc. Int'l Conf. Distributed Computing Systems (ICDCS '11), pp. 677-688, June 201

[4]Berlekamp E.R, Algebraic Coding Theory. McGraw-Hill, 1968.

[5]Garg V.K and V. Ogale, "Fusible Data Structures for Fault Tolerance," Proc. 27th Int'l Conf. Distributed Computing Systems (ICDCS '07), June 2007.

[6]HengmingZou, FarnamJahanian, "A Real-Time Primary-Backup Replication Service ", IEEE Transactions On Parallel And Distributed Systems, Vol. 10, No. 6, Pg No. 533-548, June 1999.

[7] Maria Chtepen, Filip H.A. Claeys, Bart hoedt, Filip De Turck, Piet Demeester , Peter A. Vanrolleghem , "Adaptive Task Checkpointing and Replication: Toward Efficient Fault-Tolerant Grids ", IEEE Transactions On Parallel And Distributed Systems, Vol. 20, No. 2, Pg No. 180-190, February 2009 .

[8] Ogale V, B. Balasubramanian, and V.K. Garg, "A Fusion-Based Approach for Tolerating Faults in Finite State Machines," Proc. IEEE Int'l Symp. Parallel and Distributed Processing (IPDPS '09), pp. 1-11, 2009.

[9] W.W. Peterson and E.J. Weldon, Error-Correcting Codes - Revised, second ed. The MIT Press, Mar. 1972.