ISSN (Online) 2278-1021
ISSN (Print) 2319 5940

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 4, Issue 10, October 2015

Secure Network Storage System in Distributed Applications using Fused Data Structures

Dr. S. Ravichandran

Assistant Professor, Department of Computer Science, H.H. The Rajah's College (Autonomous), Pudukkottai

Abstract: The research work describes a technique to tolerate faults in large data structures hosted on distributed servers, based on the concept of fused backups. The prevalent solution to this problem is replication. To tolerate the faults (dead/unresponsive data structures) among the whole distinct data structures, replication requires replicas of each data structure, resulting in number of servers and the number of fault for additional backups. This project present a solution, referred to as fusion that uses a combination of erasure codes and selective replication to tolerate f crash faults using just additional fused backups. This project shows that the solution achieves savings in space over replication. Further, this work present a solution to tolerate Byzantine faults (malicious data structures), that requires only backups as compared to the 2nf backups required by replication. We ensure that the overhead for normal operation in fusion is only as much as the overhead for replication. Though recovery is costly in fusion, in a system with infrequent faults, the savings in space outweighs the cost of recovery. This research work explores the theory of fused backups and provides a library of such backups for all the data structures in the Visual Studio Collection Framework. The experimental evaluation confirms that fused backups are space-efficient as compared to replication (approximately n times), while they cause very little overhead for updates. To illustrate the practical usefulness of fusion, this work use fused backups for reliability in Amazon's highly available key-value store, Dynamo. While the current replication based solution uses 300 backup structures, we present a solution that only requires 120 backup structures. This results in savings in space as well as other resources such as power.

Keywords: Data Structure, Distributed Application, Fault Tolerance, Network Storage.

I. INTRODUCTION

Distributed systems are often modeled as a set of independent servers interacting with clients through the use of messages. To efficiently store and manipulate data, these servers typically maintain large instances of data structures such as linked lists, queues and hash tables. These servers are prone to faults in which the data structures may crash, leading to a total loss in state (crash faults) or worse, they may behave in an adversarial manner, reflecting any arbitrary state, sending wrong conflicting messages to the client or other data structures (Byzantine faults).

Active replication is the prevalent solution to this problem. To tolerate f crash faults among n given data structures, replication maintains f + 1 replicas of each data structure, resulting in a total of nf backups. These replicas can also tolerate [f/2] Byzantine faults, since there is always a majority of correct copies available for each data structure.

A common example is a set of lock servers that maintain and coordinate the use of locks. Such a server maintains a list of pending requests in the form of a queue. To tolerate three crash faults among, say five independent lock servers each hosting a queue, replication requires four replicas of each queue, resulting in a total of fifteen 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.

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

II. RELATED WORK

In [1] the theory of fused state machines uses a combination of coding theory and replication to ensure efficiency as well as savings in storage messages during normal operations. Fused state machines may incur higher overhead during recovery from crash or Byzantine faults that may be acceptable if the probability of fault is low.

In [2], Fusible data structures satisfy three main properties: recovery, space constraint and efficient maintenance. The recovery property ensures that in case of a failure, the fused structure, along with the remaining original data structures, can be used to reconstruct the failed structure. The space constraint ensures that the number of nodes in the fused structures is strictly smaller than the number of nodes in the original structures. Finally, the efficient maintenance property ensures that when any of the original data structures is updated, the fused structure can be updated incrementally using local information about the update and does not need to be entirely recomputed.

In [3], Evaluation of fusion over standard benchmarks shows that efficient backups exist for many examples. To illustrate the practical use of fusion, we describe a fusion-based design of a distributed application in the Map Reduce framework. While the current replication-based solution may require 1.8 million map tasks, a fusion-based solution requires just 1.4 million map tasks with minimal overhead in terms of time as compared to replication. This can result in considerable savings in space and other computational resources such as power.

In [4], Dynamo, a highly available and scalable data store, used for storing state of a number of core services of Amazon e-commerce platform. Dynamo has provided the desired levels of availability and performance and has been successful in handling server failures, data center failures and network partitions. Dynamo is incrementally scalable and allows service owners to scale up and down based their current request load. Dynamo allows service owners to customize their storage system to meet their desired performance, durability and consistency SLAs by allowing them to tune the parameters N, R, and W.

In [5], RAIDS offer a cost effective option to meet the challenge of exponential growth m the processor and memory speeds We believe the size reduction of personal computer disks is a key to the success of disk arrays, just as Gordon Bell argues that the size reduction of micro processors is a key to the success in multiprocessors[Bell 85] In both cases the smaller size simplifies the interconnection of the many components as well as packaging and cabling While large arrays of mainframe processors (or SLEDS) are possible. it is certainly easier to construct an array from the same number of microprocessors (or PC drives) Just as Bell coined the term “multi” to distinguish a multiprocessor made from microprocessors, we use the term “RAID” to identify a disk array made from personal computer disks.

III. PROPOSED WORK

Fault tolerance is the ability of a system to perform its function correctly even in the presence of internal faults. The purpose of fault tolerance is to increase the dependability of a system. A complementary but separate approach to increasing dependability is fault prevention. This consists of techniques, such as inspection, whose intent is to eliminate the circumstances by which faults arise. In the concept of fusible data structures to maintain fault-tolerant data in distributed programs. Given a fusible data structure it is possible to combine a set of such structures into a single fused structure that is smaller than the combined size of the original structures. When any of the original data structures is updated, the fused structure can be updated incrementally using local information about the update and does not need to be entirely recomputed. In case of a failure, the fused structure, along with the correct original data structures, can be used to efficiently reconstruct the failed structure. This approach often requires significantly less space than conventional backups by replication and still allows efficient operations on the original data structures.

For example, experiments with fault servers on a distributed system suggests that for a system with k servers, this approach requires k times less space than the active replication approach.

The main benefits of replication of data can be classified as follows:

1. Performance enhancement
2. Reliability enhancement
3. Data closer to client
4. Share workload
5. Increased availability
6. Increased fault tolerance

The constraints are classified below:

1. How to keep data consistency (need to ensure a satisfactorily consistent image for clients)
2. Where to place replicas and how updates are propagated
3. Scalability

The Existing solution to this problem is replication. To tolerate the faults (dead/unresponsive data structures) among the whole distinct data structures, replication requires replicas of each data structure, resulting in number of servers and the number of fault for additional backups. Application Information Services (AIS) is replicated on different sites. It provides replication of check points on the grid. Faults ranging from machine crashes, media failures, operator errors and random data corruption results. In loss of data, both temporarily and permanently. Time Delay is very high.

Disadvantages:

- Replication of data backups
- Time Consuming
- Network Traffic was high
- High Cost (System Requirements)
- Need Additional Backups

The proposed system present a solution, referred to as fusion that uses to avoid replication. It shows that the solution achieves savings in space over replication. The fused backups are space-efficient as compared to replication (approximately n times), while they cause very little overhead for updates. In our proposed system, the data loss and time delay can be reduced when compared to the already existing services. Computer can carry pit calculation in just few seconds that would require months or perhaps even years when carried out by hand. Practically, the proposed system never makes a mistake of its own accord.

Advantages:

- Avoid Replicas
- Less Backups
- Less Processing Time
- Low Space is enough
- Network Traffic is avoided
- Low cost comparing with existing system
- Router is used for boost up the network speed

IV. METHODOLOGY USED

4.1. Insert Fused Backups

This algorithm for the insert of a key-value pair at the primaries and the backups. When the client sends an insert to a primary X_i, if the key is not already present, Xi creates a new node containing this key value, inserts it into the primary linked list (denoted primaryLinkedList) and inserts a pointer to this node at the end of the aux list (auxList). The primary sends the key, the new value to be added and the old value associated with the key to all the fused backups. Each fused backup maintains a stack (data Stack) that contains the primary elements in the coded form. On receiving the insert from X_i, if the key is not already present, the backup updates the code value of the fused node following the one contains the top-most element of X_i (pointed to by tos[i]). To maintain order information, the backup inserts a pointer to the newly updated fused node, into the index structure (indexList[i]) for X_i with the key received. A reference count (refCount) tracking the number of elements in the fused node is maintained to enable efficient deletes.

Algorithm:
- Step 1: initialize the linked list and Stack
- Step 2: Insert the backup into linked list
- Step 3: If replicas contains, insert replica data into stack
- Step 4: Get top of the stack data
- Step 5: Stored into linked list element

4.2 Delete Fused Backups

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

Algorithm:
- Step 1: Gather Top of the Stack
- Step 2: Move TOS into linked list
- Step 3: Store Linked list element
- Step 4: Clear Stack Elements
- Step 5: Set Stack is empty, Null is TOS

V. IMPLEMENTATION

5.1 Fault Tolerance

In this research work , the fault tolerance in distributed systems concept and the subjects related to this area will be discussed in a detailed manner. Firstly, some basic descriptions and concepts about fault tolerance in distributed systems will be given as a fisrt adaptation. The basic differences between faults, errors and failures will be discussed, and fault classifications will be given.

After giving the detailed information about necessary concepts, some failure models in distributed systems will be explained with some example cases.
A reliable client-server model will be explained as an example for the failure models in distributed systems. Then, main hardware reliability models, that are series and parallel models, will be mentioned in a detailed manner. After giving the models, another important issue in distributed systems will be discussed:

5.2 Replicas

Replication in computing involves sharing information so as to ensure consistency between redundant resources, such as software or hardware components, to improve reliability, fault-tolerance, or accessibility.
In this replication module, the files are received and a copy is taken. It receives the files which are sent from the different servers and it makes a copy and then the files are sent to the client. These copies are temporally stored and it does not need a memory for copying this files. After the client receives the files which are sent from different servers, this copy will be erased. So that, it doesn't need extra memory for storing that files which are sent from different servers.
Active (real-time) storage replication is usually implemented by distributing updates of a block device to several physical hard disks. This way, any file system supported by the operating system can be replicated without modification, as the file system code works on a level above the block device driver layer. It is implemented either in hardware (in a disk array controller) or in software (in a device driver).
The most basic method is disk mirroring, typical for locally-connected disks. The storage industry narrows the definitions, so mirroring is a local (short-distance) operation. A replication is extendable across a computer network, so the disks can be located in physically distant locations, and the master-slave database replication model is usually applied. The purpose of replication is to prevent damage from failures or disasters that may occur in one location, or in case such events do occur, improve the ability to recover.For replication, latency is the key factor because it determines either how far apart the sites can be or the type of replication that can be employed.

5.3 Fused Data Structure

We introduced an algorithm for the fusion of an array based stack structure. We now look at the linked list based stacks, i.e., a linked list which supports inserts and deletes at only one end, say the tail. The fused stack is basically another linked list based stack that contains k tail pointers, one for each contributing stack x_i.

When an element new Item is pushed onto stack xi, then
- If tail[i] is the last element of the fused stack, i.e, tail[i]:next = null, a new element is inserted at the end of the fused queue and tail[i] is updated.
- Otherwise, new Item is xored with tail[i]:next and tail[i] is set to tail[i]:next.

When a node is popped from a stack xi, the value of that node is read from xi and passed on to the fused stack. In

the fused stack, the node pointed to by tail[i] is xored with the old value. If tail[i] is the last node in the fused list and no other tail[j] points to tail[i], then the node corresponding to tail[i] can be safely deleted once the value of tail[i] is updated. Note that in this case, a push takes O (1) time but a pop operation may require O(k) time, since we check if any other tail points to the node being deleted. This satisfies the efficient maintenance property of fusible structures since the time required is independent of the size of the total number of nodes in the original data structure. For constant time pop operations, the algorithm for fused stacks can be applied for stacks.
The fusion of the list based stack requires no more nodes than the maximum number of nodes in any of the source stacks. The size of each node in the fused stack is the same as s, the size of the nodes in the original stack X. The only extra space overhead is the k tail pointers maintained. If all the stacks are approximately of the same size, the space required is k times less than the space required by active replication.

VI. EXPERIMENTAL RESULT

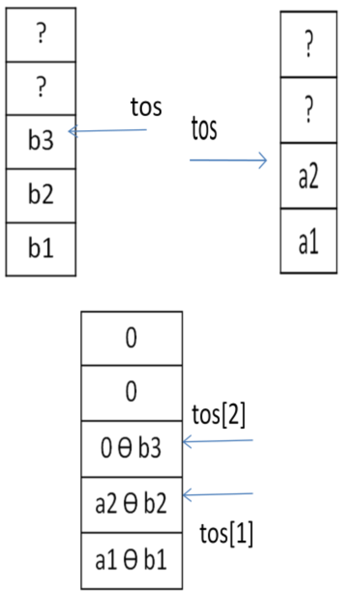

Fig 6.1 Stack Implementation

ϴ – Denotes the Fused data structures

- In this system the stack will be fused when more than one replicated data files transfer to the client machine.
- The array based stack data structure maintains an array of data, an index tos pointing to the element in the array representing the top of the stack and the usual push and pop operations.

```
Push Operation
function xi:push(newItem)
xi.array[xi.tos] := newItem;
xi.tos++;
y.push(i,newItem);
end function
function y:push(i; newItem)
y.array[y.tos[i]] := y.array[y.tos[i]] - newItem;
y.tos[i]++;
end function

Pop Operation
function xi:pop()
x.tos[i] --;
y.pop(i, xi.array[xi.tos]);
return xi.array[xi.tos]
function y:pop(i; oldItem)
y.tos[i] --;
y.array[y.tos[i]] := y.array[y.tos[i]] - oldItem;
end function
Recover Operation
function y:recover(failedP rocess)
/*Assuming that all source stacks have the same size*/
recoveredArray := new Array[y.array.size];
for j = 0 to tos[failedP rocess] ¡ 1
recItem := y[j];
foreach process p != failedP rocess
if (j < tos[p]) recItem := recItem - xp.array[j];
recoveredArray[j] := recItem;
return recoveredArray, tos[failedProcess]
```

Performance Comparison with the Existing System

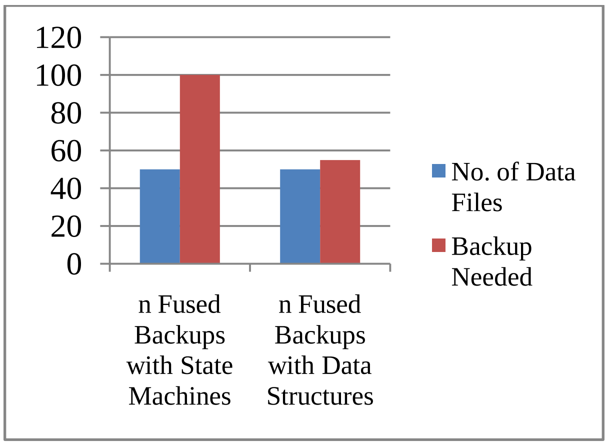

Fig 6.2 Performance Comparison with the Existing System

To correct f crash faults among n primaries, fusion requires f backup data structures as compared to the nf backup data structures required by replication. For Byzantine faults, fusion requires nf + f backups as compared to the 2nf backups required by replication.
For crash faults, the total space occupied by the fused backups in msf as compared to nmsf for replication (nf backups of size ms each). For Byzantine faults, since we maintain f copies of each primary along with f fused backups, the space complexity for fusion is nfms + msf as compared to 2nmsf for replication.

Performance of Fused Backups
This refers to the number of messages that need to be exchanged once a fault has been detected. When t crash faults are detected, in fusion, the client needs to acquire the state of all the remaining data structures. This requires n−t messages of size O(ms) each. In replication the client only needs to acquire the state of the failed copies

ISSN (Online) 2278-1021
ISSN (Print) 2319 5940

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 4, Issue 10, October 2015

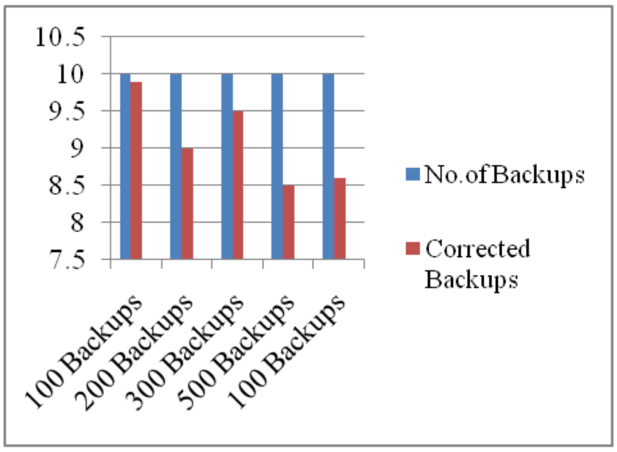

Fig 6.3 Performance of Fused Backups

Requiring only t messages of size O(ms) each. For Byzantine faults, in fusion, the state of all n + nf + f data structures (primaries and backups) needs to be acquired. This requires nf + f messages of size O(ms) each. In replication, only the state of any 2t + 1 copies of the faulty primary are needed, requiring just 2t + 1 messages of size O(ms) each.

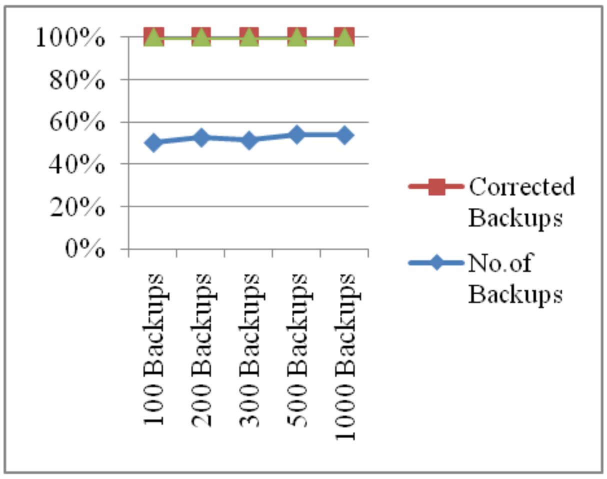

Fig 6.4 Time Complexity of Fused Backups

It defines the number of backups move from the different servers to the client also analysis the faulted and corrected backup's performance. The chart defines different backups and corrected data transfer to the client machine.

VII. CONCLUSION AND FUTURE WORK

A fusion-based technique for fault tolerance that savings in space as compared to replication with almost no overhead during normal operation. This System provide a generic design of fused backups and their implementation for all the data structures in the Visual Studio framework that includes vectors, stacks, maps, trees, and most other commonly used data structures. This System compare the main features of work with replication, both theoretically and experimentally. This work confirms that fusion is extremely space efficient while replication is efficient in terms of recovery, load on the backups and the size of the messages that need to be sent to the backups.

REFERENCES

[1] Bharath Balasubramanian and Vijay K. Garg. Fused data structure library (implemented in java 1.6). In Parallel and Distributed Systems Laboratory, http://maple.ece.utexas.edu, 2010.
[2] Vijay K. Garg. Implementing fault-tolerant services using state machines:Beyond replication. In DISC, pages 450–464, 2010.
[3] Bharath Balasubramanian and Vijay K. Garg. Fused data structures for handling multiple faults in distributed systems. In Proceedings of the 2011 31st International Conference on Distributed Computing Systems, ICDCS '11, pages 677–688, Washington, DC, USA, 2011. IEEE Computer Society.
[4] Bharath Balasubramanian and Vijay K. Garg. Fused state machines for fault tolerance in distributed systems. In Principles of Distributed Systems - 15th International Conference, OPODIS 2011, Toulouse, France, December 13-16, 2011. Proceedings, volume 7109 of Lecture Notes in Computer Science, pages 266–282. Springer, 2011.
[5] P. M. Melliar-Smith, L. E. Moser, and V. Agrawala. Broadcast protocols for distributed systems. IEEE Trans. Parallel Distrib. Syst., 1(1):17–25, January 1990.
[6] J.S. Plank and L. Xu, "Optimizing Cauchy Reed-Solomon Codes for Fault-Tolerant Network Storage Applications," Proc. IEEE Fifth Int'l Symp. Network Computing and Applications, pp. 173-180, 2006.
[7] M.O. Rabin, "Efficient Dispersal of Information for Security, Load Balancing, and Fault Tolerance," J. ACM, vol. 36, no. 2, pp. 335-348, 1989.
[8] I.S. Reed and G. Solomon, "Polynomial Codes over Certain Finite Fields," J. Soc. for Industrial and Applied Math., vol. 8, no. 2, pp. 300-304, 1960.
[9] F.B. Schneider, "Byzantine Generals in Action: Implementing Fail-Stop Processors," ACM Trans. Computer Systems, vol. 2, no. 2, pp. 145-154, 1984.
[10] F.B. Schneider, "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial," ACM Computing Surveys, vol. 22, no. 4, pp. 299-319, 1990.
[11] C.E. Shannon, "A Mathematical Theory of Communication," Bell Systems Technical J., vol. 27, pp. 379-423 and 623-656, 1948.
[12] J.K. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazie`res, S. Mitra, A. Narayanan, M. Rosenblum, S.M. Rumble, E. Stratmann, and R. Stutsman, "The Case for RAMClouds: Scalable High-Performance Storage Entirely in Dram," ACM SIGOPS Operating Systems Rev., vol. 43, pp. 92-105, 2009.
[13] D.A. Patterson, G. Gibson, and R.H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '88), pp. 109-116, 1988.
[14] W.W. Peterson and E.J. Weldon, Error-Correcting Codes - Revised, second ed. The MIT Press, Mar. 1972.
[15] J.S. Plank, "A Tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-Like Systems," Software - Practice and Experience, vol. 27, no. 9, pp. 995-1012, Sept. 1997.
[16] J.S. Plank, S. Simmerman, and C.D. Schuman, "Jerasure: A Library in C/C++ Facilitating Erasure Coding for Storage Applications - Version 1.2," Technical Report CS-08-627, Univ. of Tennessee, Aug. 2008.