Journal of Applied Sciences 13 (6): 790-799, 2013
ISSN 1812-5654 / DOI: 10.3923/jas.2013.790.799
© 2013 Asian Network for Scientific Information

A High Performance Protocol for Fault Tolerant Distributed Shared Memory (FaTP)

[1]Mutasem Alsmadi, [2,3]Usama A. Badawi and [4]Hosam E. Reffat
[1]Department of MIS, Collage of Applied Studies and Community Service, Dammam University, Dammam, P.O. Box 40287, Al-Khobar 31952, Kingdom of Saudi Arabia
[2]Computer Science Division, Department of Mathematics, Faculty of Science, Cairo University, Egypt
[3]Dammam University, P.O. Box 40287, Al-Khobar 31952, Kingdom of Saudi Arabia
[4]Department of Computer Science, El-Baha Private College of Science, Kingdom of Saudi Arabia

Abstract: In distributed environments, runtime failures often occur. If the distributed system has the ability to handle such failures dynamically (within runtime), it is said to be fault tolerant. Such systems suffer from the problem of being slow if compared to other non-fault tolerant systems. Moreover, if the system is based on a Distributed Shared Memory (DSM) in exchanging data among the distributed application members, then it is going to be slower and may be inefficient. In this study, a generic DSM based Fault Tolerance Protocol (FaTP) is introduced. FaTP is a high performance fault tolerance protocol. The proposed protocol is based on the Linda Tuple space DSM model. It introduces a compact set of DSM access primitives and supplied with a fault tolerance layer based on dynamic replication. The complexity of FaTP has been measured and its performance has been evaluated.

Key words: DSM, fault tolerance, replication, distributed algorithms

INTRODUCTION

In Computational Intensive Applications (CIA), the time needed to finish the required task is huge and hence the possibility for failures due to network partitions or machine crashes is high. Therefore, systems that enable CIAs to run on top of them must introduce a way to tolerate with runtime failures. Such failures may not only lead to affect the performance in the whole implementation processes, but also to stop the processes of the application execution. One solution for this problem is integrating a fault tolerance layer which gives a dynamic detection and recovery mechanisms (Sorin, 2009; Badawi, 2000; Gizopoulos *et al*., 2011).

A popular example of Distributed Shared Memory (DSM) models is the Linda model (Sudhakar and Ramesh, 2012; Chen *et al*., 2011; Buyya, 1999; Fiedler *et al*., 2005; Carriero and Gelernter, 1989), that permit functions to communicate via. introducing and restoring the data items, which is called tuples, into a DSM called tuple space. Any tuple in the tuple space can be accessed via. read/write operations. Examples of the read/write operations in Linda are in() operation, which is used to retrieve data from the tuple space and the out() operation, which is used to insert passive data to the tuple space.

Fault tolerance DSM systems suffer from being slow if compared to other DSM systems (Gizopoulos *et al*., 2011; Buyya, 1999; Fiedler *et al*., 2005). This slowness is due to two reasons. Firstly, their nature as DSM based systems which needs the existence of a third party and shared memory; which increases the latencies of the system. Secondly, the fault tolerance behavior of the DSM systems. A possible way to enhance the performance of such systems is to control the read/write operations performance.

Badawi (2009) has introduced a fault tolerant extension of PVM (parallel virtual machine), which is appropriate for real time applications. Where the proposed extension (TS-PVM) is based on the integrating of the TRIPS system fault tolerance mechanisms in PVM. While in this study; a generic DSM based fault tolerance protocol (FaTP) is introduced. FaTP is a high performance fault tolerance protocol, where the proposed FaTP is based on the Linda Tuple space DSM model. It introduces a compact set of DSM access primitives and supplied with a fault tolerance layer based on dynamic replication. The complexity of FaTP has been measured and its performance has been evaluated. The proposed protocol is based on dynamic replication. The read/write algorithms are introduced and analyzed. Then, the complexities of

Corresponding Author: Mutasem Alsmadi, Department of MIS, Collage of Applied Studies and Community Service, Dammam University, Dammam, Kingdom of Saudi Arabia

read/write operations algorithms are computed to measure the performance. FaTP is a generic protocol and may be applied to any DSM based system.

LINDA TUPLE SPACE MODEL

Linda model, which is introduced by Carriero and Gelernter (1989), Gelernter (1985) and Buyya (1999), has presented the tuple space concept. Tuple space is an applied of the associative shared memory paradigm attainable to all implementation processes. It consists tuples, which may be recuperation data through their contents instead of the physical addresses, through using a certain pattern-matching mechanism. The tuple space memory implementation is hidden from the user and then may be realized on a shared memory machine (Ahuja *et al*., 1986; Setz and Fischer, 1997).

Tuple space structure: If a task is sending a message to another task, it places the data in the form of tuple in the tuple space. When the receiver task is ready to receive this data, it retrieves this tuple form the tuple space. This behavior decouples the send and receives communications so that the sender task doesn't have to block until the receiver is ready to receive the transmitted data. Tuples can be active or passive. Where active tuples will consist at least one value which is not evaluated. All the values in passive tuples are evaluated. Each tuple in the tuple space contains a sequence of data elements of basic types such as integers, floats, characters and arrays (Hari, 2012).

Linda introduces six primitives to access the tuple space. The out() primitive takes a sequence of typed expressions as arguments, evaluates them, and inserts the tuple into tuple space. Similar to out(), eval() creates a tuple from its arguments, but a new process is created to evaluate the arguments. in() and rd() primitives take data types as their arguments and retrieve a tuple to match its input. The main difference between in() and rd() is that rd() reads the tuple only from the tuple space while in() takes the tuple from the tuple space. Beside in() and rd() which are blocking versions, Linda model has introduced inp() and rdp(), which are non-blocking primitives.

Generally, the available research implementations of tuple space concept distinguish between passive and active data, and therefore introduced operations for passive date such as in() and out() and an operation for active data (processes) which is eval(). In the proposed FaTP, objects are the main items in the space. where the object contains both data and code, hence there is no need to have different read and write sets of operations. Therefore, the in() and out() versions in this prototype deal with both passive and active data. Figure 1 shows the different operations supported by the modified Linda model.

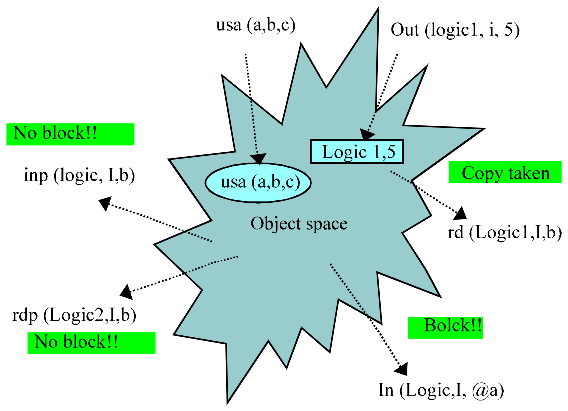

Fig. 1: Object space primitives

Tuple matching is done by creating a template from the passed pattern, which is an argument of the retrieval primitive. Tuples existing in the tuple space are then matched against this template until a matched tuple is found. Tuples and templates consist of three parts: Logic name, list of types, and list of variables. In templates, variables are separated to two types. Uppercase entries indicate formal variables; which are used in retrieval primitive to indicate "any value" for tuple fields. Lowercase letters indicate actual variables; which define a value that the matching tuple is assumed to have at the corresponding entry. In order to find a match, the template and the tuple must have equal logical names, equal cardinality, the same data types, and the same values in the corresponding actual fields.

The tuple space implements a true set of tuples, that is, it may contain several copies of identical data tuples. A given template may match several data tuples present in the tuple space. The matching algorithm is then allowed to return an arbitrary matching tuple (Setz and Fischer, 1997; Hari, 2012). Figure 1 shows an example of the matching process. The tuple (logic1, i, 5) is inserted in the tuple space using the out() primitive and has been matched and retrieved by the template (logic1, I, b) using the primitive rd().

Tuple space implementations: There are many exist implementations of the tuple space model such as the tuple space implemented by LiPS system (Library of Parallel Systems), SUN JavaSpaces system, and IBM TSpace. In this section, the LiPS tuple space is introduced as a good example of a tuple space with fault tolerance capabilities (Hari, 2012; Van Heiningen *et al*., 2006).

LiPS Tuple-Space is implemented using linked lists. It consists of three basic data structures; these are TSpace, for the tuple space, SSpace, for the sequence space and ESpace, for the event space. Figure 2 illustrates how these data structures work and hold real data tuples. The first

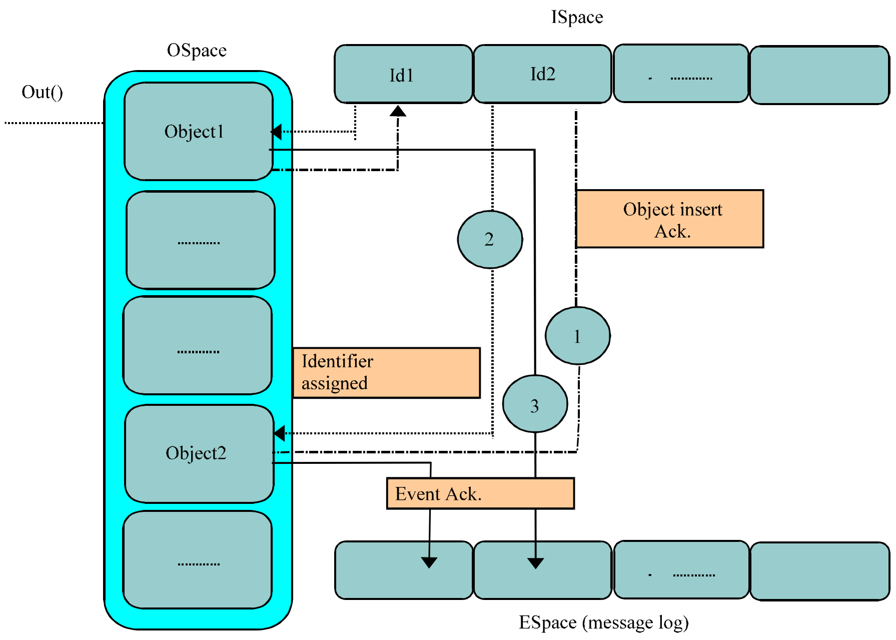

Fig. 2: An object space implementation

linked list, the TSpace, holds the tuple actual data. It is implemented as a double linked list. The SSpace puts another view on the tuple space according to the sequence numbers of tuples by which each tuple has a unique identifier in the tuple space. However, the ESpace holds the message log of the application processes in single linked list called lock. A lock contains the identifier of the application processes, its event number, the type of the operation, and the sequence number of the tuple related to that event.

The SSpace and ESpace are implemented for the recovery purpose in case of failure. The log, in ESpace, is needed in case of accessing a tuple during the recovery an application process. The sequence number of the correlated tuple is found in the event space and then the tuple is possible to be accessed. If a tuple is called by using in() or inp() operations, it is removed from the double linked lists only. If there are no more references to a tuple with a given sequence number in the event space, the tuple is first removed from the sequence space and then removed physically (Buyya, 1999).

FAULT TOLERANT PROTOCOL (FATP)

The input/output operations, such as in(), out() and eval(), play an important role in the system performance. If the system tolerates with runtime failures, then it should introduce detection and recovery protocols. The time measure of these protocols is huge if compared to the time of other parts of the system (Lazr, 2001; Liang *et al.*, 2012). In this section, the general structure of a fault tolerance DSM system is introduced as well as the insert/retrieve primitive's algorithms.

FaTP structure: The proposed protocol, FaTP, includes many layers. A group communication layer is integrated to enable message broadcasting among the distributed application members. This layer is also responsible for reporting changes in the group configuration such as machine crashes, machine join, or even network partitions. Moreover, a high availability layer has been implemented. This layer includes the detection and recovery protocols to handle different types of changes in the group of distributed processes. These two layers must be integrated in the lower level of the DSM. All types of changes, namely tuple space changes or group configuration changes must be performed through these two layers. Figure 3 shows a typical fault tolerant DSM system structure.

As shown in the figure below, an acknowledgment is sent to the high availability layer, as soon as changes will be occur in the DSM or in the group configuration.

View change handling protocol: The high availability layer constructs the View Change Handling protocol (VCH) that increases the system availability. Normally, there exist many running tuple spaces per application. Some of these spaces are active and others are passive. One of the active spaces is the original space, which is called the replica and the others are identical copies of the original space.

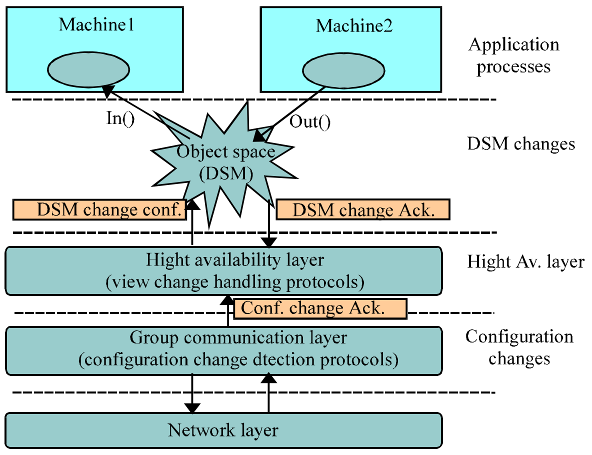

Fig. 3: A generic fault tolerant DSM system

```
Algorithm 1: View change Handling algorithm
Repeat:
    Blocking until receive a new message;
    If the received messagee is regular message then
        Move control to Object thread.
    else
        If i am the replica then
            Broadcast my object space to others.
        else if I am newly started then
            If I am the only menber in the group then
                I will be chosen as a replica for the current group
            else
                If I have an empty object space then
                    Get a copy of the object space
                else
                    Kill my self.
                end
            end
        else if I am old and there are leaving nodes then
            if replica still alive then
                I will get notification with the leaving member
            else
                Chose a new replica.
            end
        else if I am lod and there are new joined nodes then
            If I am a replica then
                Send object space copy to new members.
            else
                Move control to object space thread
            end
        end
        end
        end
    end
Until forever;
```

Fig. 4: View change handling algorithm

The VCH is responsible for spreading the effect of the client operations in all active spaces. If the client writes a tuple in the system, the protocol replicates this entry in all active spaces and ensures that all spaces are identical. Moreover, it is responsible for managing the spaces failures. It performs the client operations in the active spaces. If any active space is failed, the client will never notice system changes. The VCH failure recovery algorithm is shown in Fig. 4. This algorithm is based on the dynamic replication mechanism. Fault tolerance is

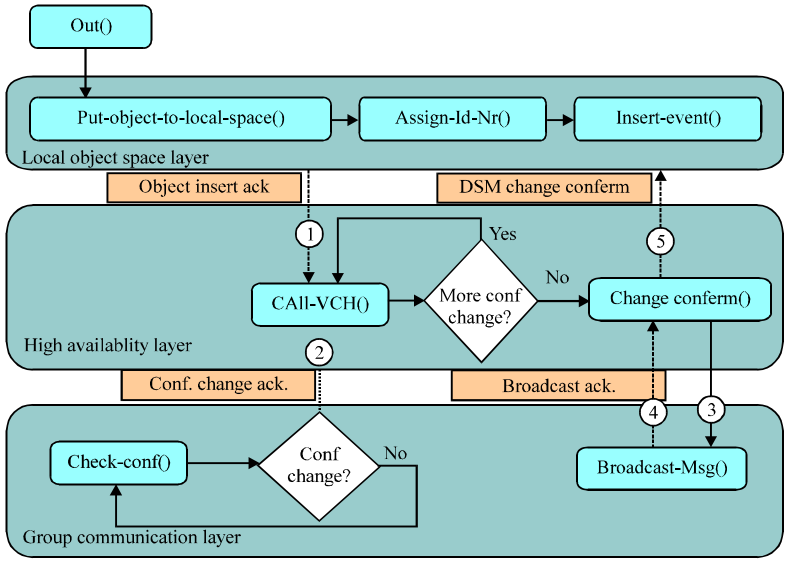

Fig. 5: Out () primitive control flow

achieved by making identical copies of the tuple space to the still alive members (current group configuration) and to make sure that these copies are identical all the time.

Input/output algorithms: In order to study the behavior of a given system, it is important to trace the functionality of its input/output primitives. In this section, the input/output algorithms for fault tolerant DSM are introduced.

Out() algorithm: Figure 5 shows the out() primitive control flow. It is clear that most of the time is elapsed in the high availability layer. This layer includes a loop that depends on the number of view changes occur. Moreover, it is clear that the tuple space will not confirm its operation unless a confirmation message arrives from the high availability layer.

As shown in the figure, the local tuple space layer includes methods to insert tuple in the tuple space, assign sequence number to the tuple, and insert the event in the event space. These steps are sequential and require minimal time. After finishing the steps in the local tuple space layer, an acknowledgment is sent to the high availability layer that includes the VCH protocol. The protocol checks for membership configuration changes; from the group communication layer, and tuple space changes, from the local tuple space layer.

The change-confirm() method is called in case there is no new configuration changes. This method sends the new tuple space view to the group communication layer to be broadcasted to all members in the group. Then, a confirmation message is sent to the local tuple space layer to commit the tuple space operation.

The in() primitive behaves in a different way. It is needed, in this case, to check the tuple space for the existence of the requested tuple. The matching mechanism is applied to match the template with existing tuples. Figure 6 shows the in() primitive control flow. As shown in the figure, the Find-Tuple() method is called as soon as the template of the required tuple is inserted to the tuple space. This method includes the matching protocol to check the existence of matching tuples. If a match found, the event is inserted in the event space and a tuple delete acknowledgment is sent to the high availability layer. Otherwise, the operation is blocked until a match exists. Similar to the out() primitive case, configuration changes, if exist, are handled. The tuple is deleted from the local tuple space physically after a confirmation message from the high availability layer is delivered. This algorithm could be applied for the rd() primitive as well.

MEASURING THE COMPLEXITY

To analyze the system performance, the complexities of the input/output primitives (out() and in()) have been measured. The recovery time has been calculated as well (Sudhakar and Ramesh, 2012).

Out() primitive complexity: The FaTP protocol broadcasts the write (out()) to all still alive active spaces. The required time to perform the write operation can be expressed by Eq. 1:

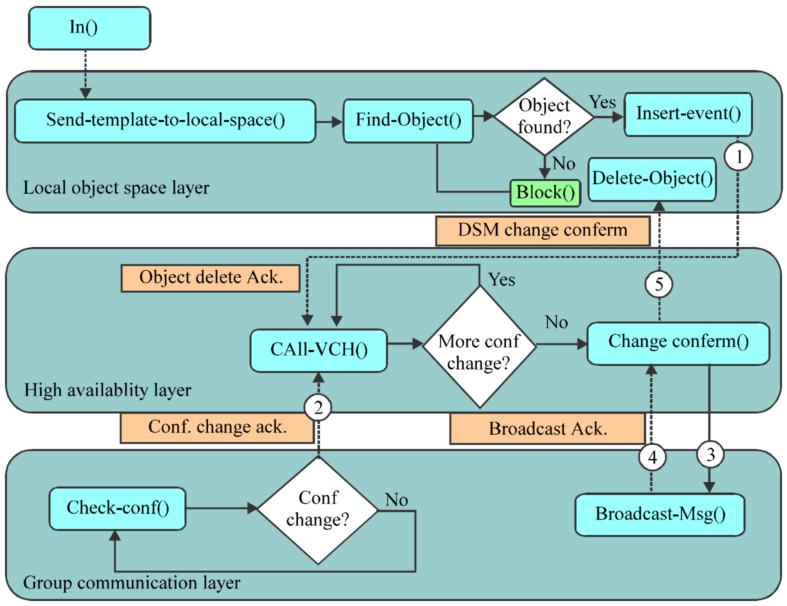

Fig. 6: In() primitive control flow

$$T_{write} = T_{cr} + T_{xp} + r \tag{1}$$

where, T_{cr} is the time required to transfer the write operation from the client to the VCH layer. T_{sp} is the time required to broadcast this operation in all active spaces. r is a summation of the time required for all active spaces in the system to perform the write operation. T_{cr} can be expressed as follows:

$$T_{cr} = \frac{E}{S_{net} \times (1 - L_{net,c})} 0 < L_{net,c} < 1 \tag{2}$$

where, E is the entry size in bytes. S_{net} is the speed of the network infrastructure. $L_{net,c}$ is the load of the network infrastructure when the client sends the write operation to the Spaces manager server.

Let:

$$K_c = \frac{1}{(1 - L_{net,c})} \tag{2a}$$

Then, T_{cr} can be expressed as follows:

$$T_{cr} = \frac{E \times K_c}{S_{net}} \tag{3}$$

Moreover, T_{sp} can be expressed as follows:

$$T_{sp} = \sum_{i=1}^{N} \frac{E}{S_{net} \times (1 - L_{net,i})} \tag{4}$$

where, $L_{net,i}$ is the load of network infrastructure during transferring the write operation from the *Spaces manager* server to the ith active space.

Let:

$$K_i = \frac{1}{(1 - L_{net,i})}$$

and:

$$K_{avr} = \sum_{i=1}^{N} \frac{K_i}{N} \tag{4a}$$

Then, T_{sp} can be expressed as follows:

$$T_{sp} = \frac{N \times E \times K_{avr}}{S_{net}} \tag{5}$$

The time required for all active spaces to perform the write operation can be calculated as follows:

$$\tau = \sum_{i=1}^{N} \tau_i \tag{6}$$

Since, all active spaces in the system are identical and the same entry will be written in all active spaces, therefore, assuming that, all active spaces take the same time to perform this write operation. According to this assumption, Eq. 6 can be written as follows:

$$\tau = N \times \tau' \quad (6a)$$

From Eq. 3, 5 and 6a, could be calculated as follows:

$$T_{write} = \frac{E \times K_c}{S_{net}} + \frac{N \times E \times K_{avr}}{S_{net}} + N + \tau' \quad (7)$$

From Eq. 7, it is noticed that, the total time to perform the write operation depends on the entry size, the number of active spaces, the network speed, the network load and the time taken to perform the write operation in the DSM. Assuming that the network speed is constant. Then, the complexity of write operation in dynamic replica is as follows:

$$O(E \times (K_c + N \times K_{avr}) + N + \tau') \quad (8)$$

In() primitive complexity: The total time to perform take (in()) operation in dynamic replica can be expressed as follows:

$$T_{take} = T_{cr} + T_{sp} + \tau_{take} + T_{jr} + T_{sc} \quad (9)$$

where, T_{cr} is the time required to transfer the take operation from the client to the dynamic replica server. T_{sp} is the time required to the dynamic replica system to execute this operation. Since, of the take operation is broadcasted in all active spaces, therefore, r_{take} equals to the summation of take operation in all active spaces and T_{jr} is summation of time to return the results from the active spaces to dynamic replica server. Also, T_{sc} is the time required to return the result to the client form the dynamic replica server. Hence, T_{cr} can be written as follows:

$$T_{cr} = \frac{E \times K_{cr}}{S_{net}} \quad (10)$$

where, K_{cr} has the same definition of K_c in (2a).

T_{sp} is summation of the transferring time of the take operation template to all active spaces. Thus, T_{sp} could be written as follows:

$$T_{sp} = \sum_{i=1}^{N} \frac{E}{S_{net} \times (1 - L_{sp,i})} \quad (11)$$

Let:

$$K_{sp,i} = \frac{1}{(1 - L_{sp,i})}$$

and:

$$K_{sp,avr} = \sum_{i=1}^{N} \frac{K_{sp,i}}{N} \quad (12)$$

Then, T_{sp} can be expressed as follows:

$$T_{sp} = \frac{N \times E \times K_{sp,avr}}{S_{net}} \quad (13)$$

The total time to perform the in() operation in all active spaces can be written as follows:

$$\tau_{take} = \sum_{i=1}^{N} \tau_i' \quad (14)$$

Since, the taken entry from all active spaces is identical and all active spaces are identical, it could be assumed that the in()operation takes the same time in all active spaces, then:

$$\tau_{take} = N \times \tau' \quad (15)$$

The time required to return the taken entries from active spaces to the Spaces manager server (T_{jr}) can be written as follows:

$$T_{jr} = \frac{N \times E \times K_{jr,avr}}{S_{net}} \quad (16)$$

Where:

$$K_{jr,avr} = \sum_{i=1}^{N} \frac{(1 - L_{jr,i})}{N} \quad (17)$$

The time required to return the result to the client from the Spaces manager server (T_{sc}) can be written as follows:

$$T_{sc} = \frac{E \times K_{sc}}{S_{net}} \quad (18)$$

Where:

$$K_{sc} = \frac{1}{S_{net} \times (1 - L_{sc})} \quad (19)$$

Then, from Eq. 13, 15, 16 and 18. Eq. 9 could be written as follows:

$$T_{take} = \frac{E \times K_{cr}}{S_{net}} + \frac{N \times E \times K_{sp,avr}}{S_{net}} + N \times \tau' + \frac{N \times E \times K_{jr,avr}}{S_{net}} + \frac{E + K_{sc}}{S_{net}} \quad (20)$$

Then, the in() operation complexity is:

$$O(N \times \tau' + N \times E \times K_{jr,avr} + E \times K_{sc}) \quad (21)$$

FaTP MEASURING AND RESULTS

Hence, practical tests are introduced to evaluate the FaTP system. First, the test environment and technique are introduced. Then, the tests and their results are presented.

Test environment and technique: The measurements are performed by using six PC's each with a CPU of type Intel Pentium 2.4 GHZ. The inter-communication among the machines is done through 100 Mbps Ethernet. Windows XP professional is the used operating system. A fault-tolerance test that is more associated to the dynamic replica is introduced. This test is based on testing the system fault-tolerance and the recovery time. Other types of tests have performed to measure the performance of the proposed protocol by testing the DSM access operations for insertion and retrieval, namely out() and in() respectively.

FaTP fault tolerance test: In this section, it is shown that the system tolerates with failures. This could be achieved by applying the following scenario. A client puts a counter in the system. It is an entry that contains an integer. The client procedure writes the entry, takes that entry, increases the counter by 1 and then rewrites the entry with the new value.

The client repeats these steps in a large number of iterations. While the client is doing this process one of the active spaces is enforced to fail. If the client process survives in spite of the failure, and the counter increases correctly, then improved that the service is fault tolerant.

Figure 7 shows the test skeleton code. The test loop is infinite. The written entry is taken to be increased and is rewritten again with the new value.

Figure 8 shows the output of the pervious test. Part (A) shows output messages of the entry counter value

```
While (true){
        Space.write(EntryCounter);
        EntryCounter = Take(tmp);
        EntryCounter.value = EntryCounter.Value +1;
        Write(EntryCounter);
        }
```

Fig. 7: Fault tolerance skeleton code test

```
A                                   B
Writing EntryCountry.value =  42    active space 1 exists
Taking EntryCountry.value =  42     active space 2 exists
Writing EntryCountry.value = 43     active space 3 exists
Taking EntryCountry.value = 43      passive space 1 exists
Writing EntryCountry.value = 44     passive space 2 exists
Taking EntryCountry.value = 44      passive space 3 exists
Writing EntryCountry.value = 45     active space 1 not-exists
Taking EntryCountry.value = 45      active space 2 exists
Writing EntryCountry.value = 46     active space 3 exists
Taking EntryCountry.value = 46      passive space 1 exists
Writing EntryCountry.value = 47     passive space 2 exists
Taking EntryCountry.value = 47      passive space 3 exists
Writing EntryCountry.value = 48     copying active space 2 to passive space 1
Taking EntryCountry.value = 48      converting passive space 1 to active space 1
Writing EntryCountry.value = 49     active space 1 exists
Taking EntryCountry.valu e = 49     active space 2 exists
Writing EntryCountry.value = 50     active space 3 exists
Taking EntryCountry.value = 50      passive space 1 not-exists
Writing EntryCountry.value = 51     passive space 2 exists
Taking EntryCountry.value = 51      passive space 3 exists
Writing EntryCountry.value = 52     active space 1 exists
Taking EntryCountry.value = 52      active space 2 exists
```

Fig. 8: Fault tolerance test results

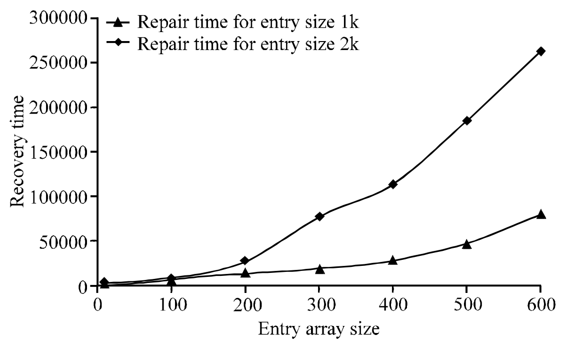

Fig. 9: Recovery time measurements in the FaTP

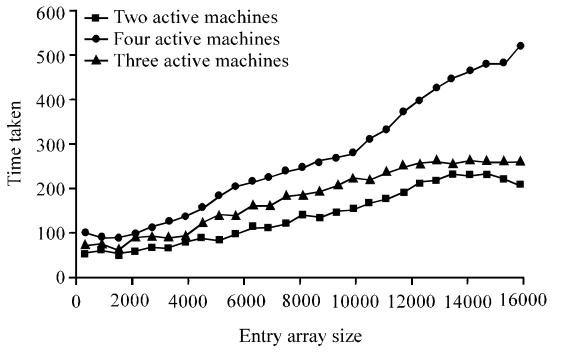

Fig. 10: Performance of out() operation in different cases (two, three and four active spaces)

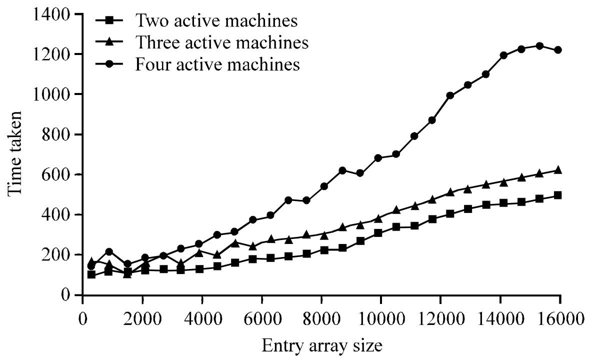

Fig. 11: Comparison between the write-take (out()-in()) operation performance for two, three and four active spaces

while writing and taking entry. The second part of the list (B) shows the tracing output messages. The output messages indicate the still alive active or passive spaces. While writing the entry that contains counter value equals 47, the first active space is enforced to fail. The *FaTP* protocol chooses passive spaces 1 to be the new active spaces. Then, the dynamic replica service copies entries from one of the still alive space (active space 2) to the passive spaces 1. Finally, FaTP service converts the passive spaces 1 to active spaces 1 and blocks the object of passive spaces 1 (not exist).

Measuring recovery time: The recovery time of the FaTP has been measured. The time taken to recover a failure in one of the active spaces equals the time required to copy the system entries from one of the still alive active spaces to one of the passive spaces plus the time required to convert the passive space to an active space. The most effective parameter in the recovery time is the number of entries in the DSM. In this test, different number of entries have been used with entry sizes 1 and 2 kbytes.

Figure 9 shows the recovery time curves. From the figure, it is noticed that the recovery time increases non-linearly by increasing the number of entries in the system. The recovery time in case of entry array size equals 2k bytes is greater than the recovery time in case of entry array size equals 1k.

Hence, the recovery time increases by increasing the entry size. In the small number of entries the recovery time curves are very close.

Performance tests: This section evaluates the effect of the number of active spaces on performance. This is done by testing the DSM access operations.

Figure 10 shows the out() operation performance in the different cases (two, three and four active spaces). This figure shows that the performance of the out() operation decreases by increasing the number of active spaces in the system. This is because the out() operation is applied in all active spaces. The difference among the three curves (two, three and four active spaces) is small at the small entry array size. The difference among the three curves (two, three and four active spaces) is small at the small entry array size. Figure 11 shows a comparison between the write-take (out()-in()) operation performance for two, three and four active spaces.

From the above figure, the four-active-spaces curve is the noisiest curve. This noise is due to the fact that increasing number of machines (active spaces) leads to communication increment. Moreover, the difference between two and three-active-space curves is smaller than the difference between three and four active space curves.

CONCLUSION

Distributed shared memory systems suffer from the problem of being slow if compared to other systems such as message passing ones. The situation would be more problematic if the system is a fault tolerant system. In this study, a high performance fault tolerance protocol, FaTP, has been presented. The proposed protocol deals with distributed shared memory based applications. It is based on the idea of integrating a high availability layer in the distributed system. Such layer is responsible for handling different changes either in the shared memory contents or in the group configuration. It guarantees the consistency of the shared memory contents among all members in the distributed application. FaTP enables the distributed application to tolerate with the failures in a reasonable timing.

FaTP structure and behavior are introduced. Moreover, the behavior of the distributed shared memory access primitives (read and write operations) is analyzed and the complexities of such primitives have been calculated. Then, the proposed protocol functionality and performance have been measured. Experimental results have shown that FaTP operates properly. It has also shown that its recovery time is reasonable.

REFERENCES

Ahuja, S., N. Carriero and D. Gelernter, 1986. Linda and friends. IEEE Comput., 19: 26-34.

Badawi, U., 2000. A single system image supporting distributed objects. Ph.D. Thesis, Department of Mathematics, Faculty of Science, Cairo University, Egypt.

Badawi, U., 2009. TS-PVM: A fault tolerant PVM extension for real time applications. Int. Arab J. Inf. Technol., 6: 424-430.

Buyya, R., 1999. High Performance Cluster Computing: Programming and Applications. Vol. 2, Prentice Hall PTR, USA., ISBN-13: 9780130137852, Pages: 664.

Carriero, N. and D. Gelernter, 1989. How to write parallel programs: A guide to the perplexed. ACM Comput. Surv., 21: 323-357.

Chen, X., Z. Lu, A. Jantsch and S. Chen, 2011. Supporting distributed shared memory on multi-core network-on-chips using a dual microcoded controller. Proceedings of the Conference on Design, Automation and Test in Europe, March 8-12, 2010, Belgium, China, pp: 39-44.

Fiedler, D., K. Walcott, T. Richardson, G.M. Kapfhammer, A. Amer and P.K. Chrysanthis, 2005. Towards the measurement of tuple space performance. ACM SIGMETRICS Perform. Eval. Rev., 33: 51-62.

Gelernter, D., 1985. Generative communication in linda. ACM Trans. Programm. Languages Syst., 7: 80-112.

Gizopoulos, D., M. Psarakis, S.V. Adve, P. Ramachandran and S.K.S. Hari *et al*., 2011. Architectures for online error detection and recovery in multicore processors. Proceeding of the Design, Automation and Test in Europe, March 14-18, 2011, Grenoble, France.

Hari, H., 2012. Tuple space in the cloud. Master's Thesis, Department of Information Technology, Uppsala University, Sweden.

Lazr, I., 2001. Designing a fault-tolerant jini compute server.http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.19.3916

Liang, G., B. Sommer and N. Vaidya, 2012. Experimental performance comparison of Byzantine fault-tolerant protocols for data centers. Proceedings IEEE INFOCOM, March 25-30, 2012, USA., pp:1422-1430.

Setz, T. and J. Fischer, 1997. Fault-tolerant distributed application in LIPS. Technical Report SFB 124 09/1997, University of the saarland at Saarbrucken, Germany.

Sorin, D.J., 2009. Fault Tolerant Computer Architecture. Morgan and Claypool Publishers, USA., ISBN: 9781598299533, Pages: 103.

Sudhakar, C. and T. Ramesh, 2012. An improved DSM system design and implementation. Int. J. Next-Generation Comput., Vol. 3.

Van Heiningen, W., T. Brecht and S. MacDonald, 2006. Babylon v2.0: Middleware for distributed, parallel and mobile Java applications. Proceedings of the 11th International Workshop on High-Level Parallel Programming Models and Supportive Environments, April, 2006, Rhodes Island, Greece.