High Available Fault Tolerant Technique in Distributed Shared Memory

Hosam E. Refaat
Dept. of Information Systems
in Suez Canal University, Egypt.

Usama Badawi
Dept. of IT in Damam
university, KSA.

ABSTRACT
distributed systems, that are based on constructing a network of heterogeneous computers, suffer from the problem of failing components during the system run time. In case of failure, the distributed applications must be restarted from the scratch. The main goal of this research is to add the dynamic failure recovery technique to the JavaSpaces server. So, the client continues its jobs while failures occur in the system. Also, the new technique in JavaSpaces is evaluated by analyzing and testing.

General Terms
Parallel systems, Distributed shared memory, Fault Tolerant.

Keywords
Parallel systems, Distributed shared memory, Fault Tolerant, Linda system, Tuple-space, Jini, JavaSpace.

1. INTRODUCTION
Nowadays, high speed networks and microprocessors making clusters of workstations are appealing vehicles for effective parallel computation. In other words, clusters redefine the concept of supercomputer. Parallel systems can be classified in different ways. The most famous classification depends on the Memory-Access strategies. In the Memory-Access classification, there are three types of parallel systems, namely; distributed memory, shared memory, and distributed shared memory. In distributed memory systems, each node in the system has its private memory. If any node needs data from another, it will send a request message to it. Hence this system is also called *"message passing"*. Parallel Virtual Machine (PVM) [1] is one of the most commonly used parallel systems that are based on the message passing concepts. In the message passing systems, if any node needs to send a message to another node, it must know the receiver node address. The direct connection between nodes will increase the performance but the parallel application will depend on the system structure (machine addresses)[2, 3].

The second type is *"shared memory"*. The shared memory systems are based on the existence of a global memory shared among all nodes in the system. A shared memory system has various advantages, these are; it is simple, it eases the data sharing and it eases the implementation of the parallel application. If any node needs to send a message to another node, it sends the message to the shared memory and the receiver node takes it from the shared memory. Also, this structure requires a high cost of communication. The time of sending a message between two nodes is approximately duplicated if compared with the message passing systems, because of the existence of third part (the shared memory) [4].

The *"Distributed Shared Memory"* (DSM) takes the advantages of the previous two types. DSM is based on constructing a shared memory with a low cost. This is done by constructing a virtual shared memory using the available distributed memories. Thus, DSM acts as a physical shared memory. The DSM systems have some countable advantages over the message passing based ones, these are; the application level ease of use, the distributed system is portable, and it is easy to share data and processes. Since the sender does not need to know the address of the receiver, then the structure of the application becomes simpler and the application code is more readable. Also, in DSM based systems, it is easy to share data and processes among the nodes by inserting the data or processes in the shared memory [5, 6].

During the distributed system is run, some problems may occur such as, machine failure or network partitioning. These problems can cause the application to be stopped (system failure). The failure in intensive computation systems may cause losing data that must remain to survive the application. These data is called *"persistent data"*. The only solution for losing persistent data problems is to restart the system from scratch. With these systems, we can't be sure that the system will take a desirable time and finishes correctly. These systems are called *"unreliable systems"*. In order to have a *"reliable system"*, these problems must be solved. There are three strategies to solve these problems, these are; prevent failure, reduce the failure and tolerate with the failure. The first strategy is to prevent the failure during runtime by careful system design. This strategy is not applicable. The second strategy is to allow failure and maximize the period of time between failures. Again this strategy is not applicable because it can't prevent the failure occurrence in a certain period of time. The third strategy is to tolerate with the failure and to handle the failure dynamically. Systems that use the third strategy are called *"fault-tolerance systems"*. There are many fault-tolerance protocols to handle the failure. Some of these protocols are to handle the client failures, like transaction and mobile co-ordination. And some of them are to handle the server failures, like replication. Moreover, it is important to handle the failure dynamically and maximize the percentage of the time available for productive use (*system availability*) to have a *high available system* [6, 7, 8, 9].

This paper introduce a method for integrate a fault tolerant technique in DSM, which is based on create a dynamic hot replicas. This technique is implemented in DSM service over Jini system, which is called "JavaSpace". The next section introduce the JavaSpace service. The dynamic hot replicas is introduced in Section 3. The last section discusses the performance result.

2. JINI- JAVASPACE SYSTEM
Jini system extends the Java environment from a single virtual machine to a network of virtual machines. Jini system is a distributed system based on the idea of federating a group of users and the resources required by these users to have a large monolithic system [8]. The power of the Jini comes from the services, since services can be anything joined to the network.

A JavaSpace is a service in Jini system that implements the DSMS model.

JavaSpace is a distributed shared memory service that is implemented over Jini System [10]. The object that can be written in JavaSpace service is called an *"entry"*. The entry contains data or/and processes. Sometimes the entry is called tuple. JavaSpace contains the following operations: take, takeIfExists, read, readIfExists, write, notify, snap¬shot. The write operation is to write an entry in JavaS¬pace. To read an entry from the JavaSpace, the read() or readIfExists() operation is used. The consecutive reading operation of the same template may return different entries even if JavaSpace contents are not changed. The difference between these two versions of reading is that; readIfExists() is not blocked if the tuple is not found in the space, it returns a null tuple if there is no matching tuple. Take() or takeIfExists() are two operations that extract entries from JavaSpace. In other words, these operations are similar to read and readIfExists() operations except that; taking operations remove the entry from the space. The snapshot operation is to take a copy of existing entry, but this copy is not updated in spite of the changes that may occur in the original entry. The notify operation is used to define an event that triggers when a specific entry is written [10].

3. DYNAMIC HOT REPLICA

The idea behind the dynamic replica system is to construct a new layer that increases the system availability. This new layer is called *"SpacesManager"* layer. In our System, there are many JavaSpaces services, some of these spaces are active and some of them are passive, as seen in Figure 1. All tuple space operations are performed on active spaces. One of the active spaces is the original tuple space, which is called the replica, and the others are identical copies of the original space. The SpacesManager layer is responsible for spreading the effect of the client operations in all active spaces.

If the client tries to write an entry in the dynamic replica system, the SpacesManager replicates this entry in all active spaces and insures that all spaces are identical. Also, if the client takes an entry, this entry will never be seen in all active spaces until it is rewritten. There is no need to replicate the effect of read operation in all active spaces, because it will be a redundant operation. Since all active spaces are identical, it will be enough to read the entry from the original space or any still alive active space.

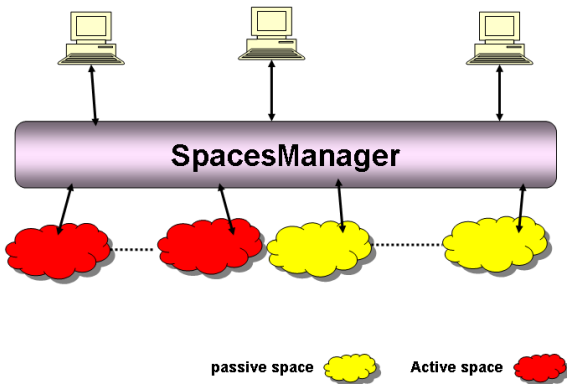


Fig 1: High Availability Layer in Javaspaces

The take operation must be spread in all active spaces. A typical problem may raise here, the same take operation can return different results for each trial. This is because of the possibility of having entries with similar attribute values in different spaces. This problem can be solved by performing the take operation in the original space at first. Then the ID of the taken entry will be used in the take template to delete this special entry from the other active spaces. By this way we can maintain the system spaces identical. Moreover, to be sure that all spaces are consistent, any operation spreads in all active spaces is done in all spaces under the same transaction. On the other hand, the SpacesManager layer is responsible for managing the spaces failures. It is responsible for performing the client operations in the active spaces without making the client aware of the details. Figure 1 shows SpacesManager that acts as a high availability layer. The client in this system is not aware of the JavaSpaces location or the number of JavaSpaces that it deals with. Thus, if any one of the active spaces is failed, the client will never notice system changes because this event will be handled by the SpacesManager failure recovery protocol. The SpacesManager failure recovery algorithm is shown in Figure 2.

The algorithm handles different failure types depending on the type of failed machine. If the failed machine contains an active space, the response depends on whether the failed active space is original or not. If the failed machine is the original space, one of still alive active spaces is chosen to be the original space. There is no difference between the original machine and the other active machines. To survive a number of active spaces from perishing, one of the passive spaces is initiated and inserted in the list of active machines. The new active space receives a copy of all entries. Also, if any of the active spaces other than the original space is failed, one of passive spaces is chosen to be the new active space and it receives a copy of all entries. In case of machine failure or network partition occurs, the dynamic-replica system blocks this machine. In other words, the dynamic-replica system will delete this machine from the active spaces list. The dynamic-replica system blocks any failed machines whether it is active or passive. If the failed/disjoined machine comes back to the system, the dynamic-replica system deletes all entries in its JavaSpaces and rejoins it as a passive machine.

To have more technical view of the system, the class diagram for the *SpacesManager* layer is shown in Figure 3. The SpacesManager service is based on defining a new tuple space control service in the Java RMI. Thus, the SpacesManager interface contains the basic tuple space operations (write, take and read). This interface extends the Java API Remote interface. The "SpacesManager" interface is an abstraction of the service.

Algorithm : SpacesManager failure recovery algorithm

Initialization;
Check the existing JavaSpaces;
Choose the active spaces and passive spaces;
One of the active spaces will be the original and the others are replicas;
repeat
  Check the existing JavaSpaces;
  if *machine Failure occurs* **then**
    if *one of the active spaces is failed* **then**
      if *the failed space is the original* **then**
        Block this machine;
        One of the active spaces is chosen to be the original space;
        One of the passive spaces is chosen to be active;
        Clear all entries in the new active space;
        Copy all entries from any of still alive active space to the new active space;
      if *If the failed space is not the original (replica)* **then**
        Block this machine;
        One of the passive spaces is chosen to be active;
        Clear all entries in the new active space;
        Copy all entries from any of still alive active space to the new active space;
      if *If the failed space is passive space* **then**
        Block this machine;
  if *One of failed machine comes back to life* **then**
    Delete all entries in its space;
    Join this machine as a passive machine;
    Unblock this machine;
until *no condition*;

Fig 2: SpaceManager Algorithm

On the other hand, the "*SpacesManagerImp*" is a class that implements the SpacesManager interface and extends the java API interface *UnicastRemoteObject*. This class calls the "getSpacesThread" thread in its constructor. The getSpacesThread is a thread that contains an infinite loop to check the still alive JavaSpaces. This checking mechanism for the existing spaces is repeated periodically. The getSpacesThread class contains a public variable of type vector called "SpacesObject". The SpacesObject vector contains objects of all JavaSpaces services in the system and another metadata like the type of space (active or passive), block...ect. The setSpacesThread is responsible for managing the failure. It uses the checkSpaces method to check the existence of the system machines. The checkSpaces method calls the JSServiceLocator class to check the existence of the JavaSpaces service. It uses the convertSpace method to convert the passive spaces to active spaces and the copySpace method to copy all entries from one of still alive active spaces to the new active space. The setSpacesThread uses the flushSpace method to delete all entries from the rejoining machine. The "SpacesManagerClient" is a client program that is used to test our service using our resizable entry "MyEntry". The client program fetches the dynamic replica service using the "ServiceLocator" class. On the other hand, the client code uses this service using its proxy class called "SpacesManagerInfProx". This proxy allows the user of add some code in the service operations.

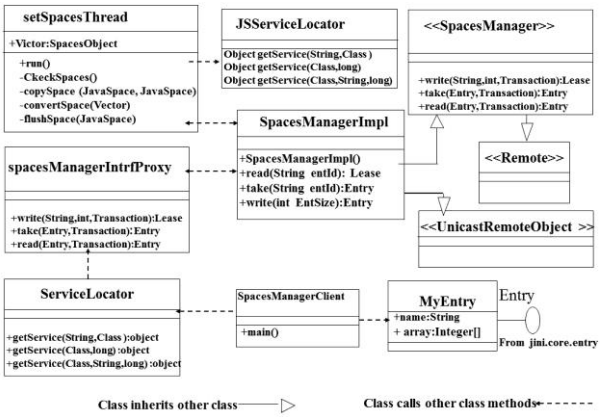


Fig 3: Class diagram of the Dynamic Hot Replica

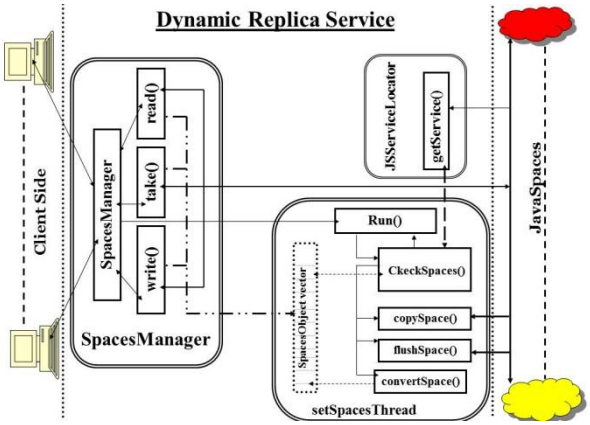


Fig 4: Flow control of the Dynamic Hot Replica

4. SYSTEM COMPLEXITY

This section measures the complexity of the dynamic replica system. This is to be done by measuring the complexity of the dynamic replica operations and the complexity of the recovery time. To analyze the system performance we focus on the methods that perform the user operations and the system recovery method. In other words, the Linda system operations (read, write and take) and the recovery method is analyzed [11]. The discussion starts with the write operation complexity. The dynamic replica service performs the client write operation by broadcasting this operation in all still alive active spaces. The total time to perform the write operation in the dynamic replica system is shown in the following equation:

$$T_{write} = T_{cr} + T_{sp} + \tau$$

Where, T_{cr} is the time required to transfer the write operation from the client to the dynamic replica server. T_{sp} is the time required for the dynamic replica system to broadcast this operation in all active spaces. τ is a summation of the time required for all active JavaSpaces in the system to perform the write operation. T_{cr} can be written as follows:-

$$T_{cr} = \frac{E}{S_{net} * (1 - L_{net,c})}$$

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 SpacesManager server, also $0 < L_{net,c} < 1$. Let

$$k_c = \frac{1}{1 - L_{net,c}}$$

Thus T_{cr} can be written as follows:-

$$T_{cr} = \frac{E * k_c}{S_{net}}$$

Similarly, T_{sp} can be written as follows:-

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

Where, $L_{net,i}$ is the load of network infrastructure during transferring the write operation from the SpacesManager server to the active spaces number $"i"$. Also suppose that, $k_i = \frac{1}{1 - L_{net,i}}$. So, the previous equation can be written as follows.

$$T_{sp} = \sum_{i-1}^{N} \frac{E * k_i}{S_{net}}$$

Let $k_{avr} = \sum_{i=1}^{N} \frac{1}{1 - L_{net,i}}$, thus T_{sp} can be written as follows:

$$T_{sp} = \frac{N * E * k_{avr}}{S_{net}}$$

Also, the summation of the time required to all active spaces to perform the write operation is as follows:-

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

Since all active spaces in the system are identical and the same entry will be written in all active spaces, therefore, assume that; all active spaces take the same time to perform this write operation (τ'). In this case, the previous equation can be written as follows.

$$\tau = N * \tau'$$

So, the T_{write} can be written as follow:

$$T_{write} = \frac{E * k_c}{S_{net}} + \frac{N * E * k_{avr}}{S_{net}} + N * \tau'$$

From the previous equation notice 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 write operation in JavaSpaces. the network speed is assumed to be constant. Thus, the complexity of write operation in dynamic replica is

$$O(E * (k_c + N * k_{avr}) + N * \tau')$$

Now, the complexity of the read operation is measured.

$$T_{read} = T_{cr} + T_{sp} + \tau_{read} + T_{jr} + T_{sc}$$

Where, T_{cr} is the time required to transfer the read operation from the client to the dynamic replica server. T_{sp} is the time required to the dynamic replica system to demand execution of this operation from one of active spaces. τ_{read} is required to JavaSpaces to perform the read operation. T_{jr} is the time required to return the result form the active space to the dynamic replica server. And T_{sc} is the time required to return the result to the client form the dynamic replica server. T_{cr} can be written as follows.

$$T_{cr} = \frac{E_o}{S_{net}(1 - L_{net,cr})}$$

Where, E_o is the size of read template in bytes. $L_{net,cr}$ is the network load when the client transfers the read template to the SpacesManager server. Now, suppose that $k_{cr} = \frac{1}{1 - L_{net,cr}}$

thus T_{cr} is written as follows.

$$T_{cr} = \frac{E_o * k_{cr}}{S_{net}}$$

Since the dynamic replica server performs the read operation for once in any of the still alive active spaces, then L_{sp} can be written as follows.

$$T_{sp} = \frac{E_o}{S_{net}(1 - L_{net,sp})}$$

Where, $L_{net,sp}$ is the network load when the SpacesManager transfers the read template to one of the active JavaSpaces. Also, suppose that $k_{sp} = \frac{1}{1 - L_{net,s}}$. Thus, T_{sp} is written as follows.

$$T_{sp} = \frac{E_o * k_{sp}}{S_{net}}$$

Similarly, the time required to return the result back is shown in the following equations.

$$T_{jr} = \frac{E_r}{S_{net}(1 - L_{net,jr})}$$

$$T_{sc} = \frac{E_r}{S_{net}(1 - L_{net,sc})}$$

Where, E_r is the size of result entry, $L_{net,jr}$ is the network load while returning the result entry to the SpacesManager server and $L_{net,sc}$ is the network load while returning the result to the client. Also, suppose $k_{jr} = \frac{1}{1 - L_{net,jr}}$, $k_{sc} = \frac{1}{1 - L_{net,sc}}$ then the previous two equations will be written as follows.

$$T_{jr} = \frac{E_r}{S_{net}(1-L_{net,jr})}$$

$$T_{sc} = \frac{E_r}{S_{net}(1-L_{net,sc})}$$

Then the total time required to perform the read operation in the dynamic replica is shown in the following equations.

$$T_{read} = \frac{E_o * k_{cr}}{S_{net}} + \frac{E_o * k_{sp}}{S_{net}} + \tau_{read} + \frac{E_r * k_{sc}}{S_{net}} + \frac{E_r * k_{sc}}{S_{net}}$$

From the previous equation the total read time depends on the size of read operation template, JavaSpaces reading time, the size of result entry and the network load while the operation steps. The size of the read operation template can be neglected, because it is almost very small. Thus, the complexity of read operation in the dynamic replica is

$$O(E_r * (k_{jr} + k_{sc}) + \tau_{read})$$

Similarly, the time of take operation can be computed as the following.

$$T_{take} = T_{cr} + T_{sp} + \tau_{take} + T_{jr} + T_{sc}$$

Because of the take operation is broadcasted in all active spaces, then T_{sp} is summation of time for transferring the take operation template to all active spaces. Thus it can be written as follows.

$$T_{sp} = \sum_{i=1}^{N} \frac{E_o}{S_{net} * (1-L_{sp,i})}$$

Where, $L_{sp,i}$ is the network load while transferring the take operation template from the SpacesManager to the active space "i". Suppose, $k_{sp.i} = \frac{1}{1-L_{sp,i}}$. Thus, the previous equation can be written as follows.

$$T_{sp} = \sum_{i=1}^{N} \frac{E_o * k_{sp,i}}{S_{net}}$$

Suppose, $k_{sp.avr} = \sum_{i=1}^{N} \frac{k_{sp.i}}{N}$ then the previous equation will be written as follows.

$$T_{sp} = \frac{E_o * k_{sp,avr}}{S_{net}}$$

Also, the total time for performing the take operation in all active spaces can be written as follows.

$$\tau_{take} = \sum_{i=1}^{N} \tau'_i$$

Since the taken entry from all active spaces is identical and all active spaces are identical, thus we can suppose that the take operation takes the same time in all active spaces (τ'). So, τ_{take} can be written as follows.

$$\tau_{take} = N * \tau'$$

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

$$T_{jr} = \sum_{i=1}^{N} \frac{E_r}{S_{net}(1-L_{jr,i})}$$

Where, $L_{jr,i}$ is the network load when returning the result to the SpacesManager server. E_r is the size of result entry. Similarly, suppose that $k_{jr,i} = \frac{1}{1-L_{jr,i}}$ and $k_{jr,avr} = \sum_{i=1}^{N} \frac{E_{jr,i}}{N}$. So, T_{jr} can be written as follows.

$$T_{jr} = \frac{N * E_r * k_{jr,avr}}{S_{net}}$$

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

$$T_{sc} = \frac{E_r * k_{sc}}{S_{net}}$$

Where, $k_{sc} = \frac{1}{S_{net} * (1-L_{sc})}$ is defined previously. Since, L_{sc} is the load of the network while returning the result to the client. Thus, the previous equation can be written as follows.

$$T_{take} = \frac{E_o * k_{cr}}{S_{net}} + \frac{E_o * N * k_{sp,avr}}{S_{net}} + N * \tau' + \frac{N * E_r * k_{jr,avr}}{S_{net}} + \frac{E_r * k_{sc}}{S_{net}}$$

Similarly, the complexity of the take operation in the dynamic replica server is

$$O(N * \tau' + N * E_r * k_{jr,avr} + E_r * k_{sc})$$

Now it is time to measure the complexity of the system recovery process. The recovery process is based on copying all entries from one of still alive active spaces to the new active space. Thus, the total recovery time for the failure can be defined as follows.

$$T_{\text{Recovery}} = \sum_{i=1}^{\mu} (T_{read,i} + T_{write,i})$$

Where, μ is the total number of entries in any one of still alive active space. $T_{read,i}$ is the time taken to read entry number "i". Also, $T_{write,i}$ is the required time to write entry number "i" in the new active space. Thus, the complexity of recovery operation in dynamic replica is.

$$O(\mu * (\tau_{read} + E_r * k_{rjs,avr} + E_r * k_{wsj,avr} + \tau_{write}))$$

5. SYSTEM PERFORMANCE TEST

The measurements in the dynamic replica systems are performed by using six PC's. These PCs have a CPU of type Intel Pentium 2.4 G.H and 512 RAM. The intercommunication among the machines is done by 100 Mbps Ethernet. The software environment includes Windows XP professional as an operating system, Java JDK 1.4.2_04[8], Jini(TM) Technology Starter Kit v2.0.2 [9] and a free visual platform for Jini 2.0 that is calledblitz-javaspace. Inca X(TM) [12].

The performance measurements of dynamic replica system are based on the send and send-receive operations, which are tested here [13,14]. The test operations are as follows; write, write-read and write-take operations. All tests have been repeated for 100 iterations and the average is calculated. The practical tests are done on two, three and four active spaces dynamic replica systems. Two test strategies is used on each dynamic replica system, the first test strategy measures the entry size effect. This test strategy is done by using a resizable entry. In the second strategy the distribution time of system operations is measured. This test is done by repeating each tuple space primitive in different number of iterations (begin from 100 to 1000 iterations). This test is shown in the following subsection.

There is a third type of tests that is more associated to the dynamic replica. It is called "fault-tolerance test". This test is based on testing the system fault-tolerance and the recovery time. The fault-tolerance test is seen in Section 5.2. Finally, the effect of number of active spaces on system is measured in Section 5.3.

5.1 System Performance measurement

Now, the performance of dynamic replica system that has four active spaces using variant entry size is tested. Figure 5 shows the effect of changing the entry size in the performance of the four active spaces dynamic replica system.

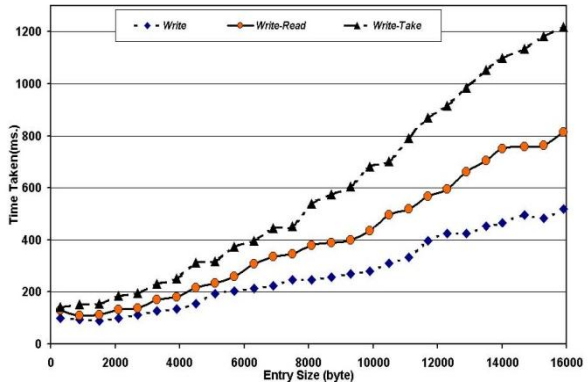


Fig 5: The Dynamic-Replica Entry-Size Test in Case of Four Active Machines

From Figure 5 we can notice that; there is a little noise in the three curves. The changes in the network load could be the reason of this noise. In the small entry size the difference among the three operations is small. In other words, the three curves are very close in the small entry size. Also by increasing the entry size, the time taken for each operation increases and the difference among the curves increases. All curves increase non-linearly.

Our observations on this figure also are logic except for some noisy points in which the network load or machine load takes place. It is normal to see that the time to treat the entry increases as the entry size increases.

Now the number of operations effect on the dynamic replica in case of two active spaces is tested. This test is done using different sizes of entries (100, 500, 1000 Bytes in the Entry Array Size)

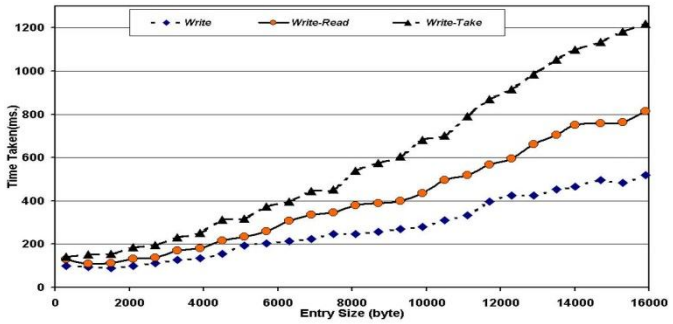


Fig 6: Timing of Dynamic Replica operations against the number of entries when entry array size is 500 bytes in case of four active machines

Figure 6 shows the performance of four active spaces dynamic replica system using an entry contains an array of size 500 bytes in its array. In the figure, we can notice that, at the point 100 operations the write curve is near to write-read curve. The write curve at point 100 operations has a time 11143 ms. and the write-read curve takes 11578 ms. The three curves are close at the first test point (100 operations). Also, the three curves increase linearly until the point 300 operations.

The difference between the write-read curve and the write-take curve is greater than the difference between the write curve and the write-read curve. All curves in this figure increase non-linearly.

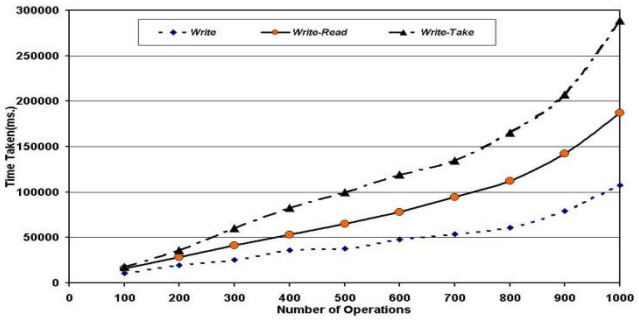


Fig 7: Timing of Dynamic Replica operations against the number of entries when entry array size is 1000 bytes in case of four active machines

Figure 7 shows the curves that represent the relation between number of operations per second test using an entry that contains 1000 bytes in its array. From this figure we can notice that; the three curves are very close at the first test point (100 operations). The write-read and write-take curves are very close at points 100 and 200 operations. Also, the write-read curve has approximately a linear behavior till 600 operations. Moreover, the write curve increases linearly until the test point 400 operations. In general all curves are increased non-linearly.

Again, our observations on this set of figures are logic except for some noisy points in which the network load or machine load takes place. It is normal to see that the time to treat the entry increases as the number of operations performed is higher for a fixed entry size. In general, the write-take operation takes the greatest time and the write operation takes the smallest time. Both the entry size tests and the number of operations tests give non-linear curves for all operations. At the small number of operations the differences in the

performance of the write, write-read and write-take operations is very small.

5.2 Dynamic Replica Highly Available Layer Test

In this section answers the following questions. Is the system fault tolerant? If yes, is the system actually highly available? What is the recovery time?

The answer of the first question could be achieved by making the following test. A client will put a counter in the system. This counter is an entry that contains an integer. The client procedure is writing the entry, taking that entry, increasing the counter by 1 and then rewriting 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. The question now is; will the client process survive in spite of the failure? And does the client counter increase correctly? If the answer to these two questions is "yes" this proves that, the system is fault tolerant.

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

Fig 8: Fault tolerance skeleton code test

Figure 8 shows a skeleton code for the test steps. In this figure the test loop is infinite. The written entry is taken to be increased and is rewritten again with the new value. Figure 9 shows the output of the pervious test. Part (A) of this figure shows output messages of the entry counter value while writing and taking entry. The second part of the figure (B) shows the setSpacesThread output messages. The output messages indicate the still alive active or passive spaces. While the counter loop is repeated infinitely, there is a failure in the first active JavaSpaces. While writing the entry that contains counter value equals 47, the first active JavaSpaces fails. The dynamic replica service chooses passive spaces1 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 spaces1. Finally, dynamic replica service converts the passive spaces1 to active spaces1 and blocks the object of passive spaces1 (not exist).

In Figure 9.A notice that, while the failure happens the counter value continues to increase. The client continues its operation and the counter values are sequential. The reason for this is that, the dynamic replica is responsible for broadcasting the client operations in all active spaces. If there is a failure in one or more of active spaces, it means that there is still alive one or more of active spaces that contain the data. Also, in case of failure, one or more of passive spaces will be converted automatically and each one of the new active spaces has a copy of the data. In another words there is no losing in the data.

To answer the second question, we make another test which is like the pervious one. In this test there are many dynamic replica services and the client will do the previous process. In other words, the client creates a counter and increases it as the previous test. Moreover, the client takes a new instance of the dynamic replica service before doing any operation (write, read, take operations). While the client is doing this process, the dynamic replica service (which the client connected with it) is enforced to fail. We will find that, the new dynamic replica service continues the client process successfully. This is because the client in each operation takes a new instance of any one of the still alive dynamic replica services and each dynamic replica service monitors all JavaSpaces in the system.

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

Fig 9: The output of the fault tolerance skeleton code test

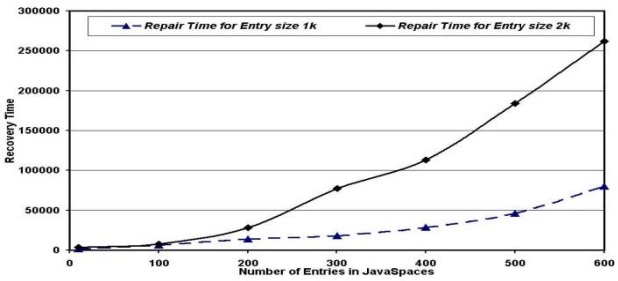


Fig 10: Dynamic replica recovery time

Finally, the recovery time of the dynamic replica-fault tolerance method is 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 dynamic replica system. This test measures the recovery time of our system by different number of entries using the most common entry size (1,2k bytes in the entry array size). Figure 10 shows the recovery time for the dynamic replica fault-tolerance method. Notice from this figure, the recovery time increases non-linearly by increasing the number of entries in the system. Also, the repair time in case of entry array size equals 2k bytes is grater than the repair time in case of entry array size equals 1k. So, the recovery time increases by increasing the entry size. In the small number of entries the recovery time for 1 k byte curve is very close to 2k byte curve. This test can't be done in the transaction or mobile coordination fault-tolerance methods because these fault-tolerance methods are client-side fault-tolerance type.

5.3 Results Comparison

This section evaluates the effect of the number of active spaces on the dynamic replica system. Now, the pervious test results of the two, three and four active spaces dynamic replica systems is compared.

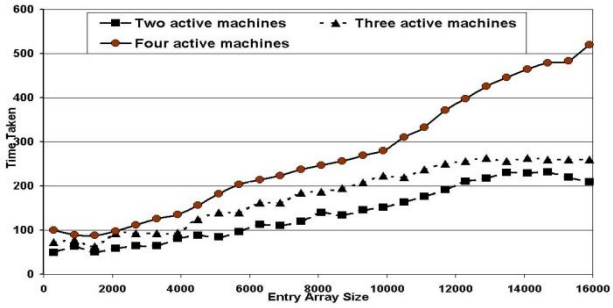
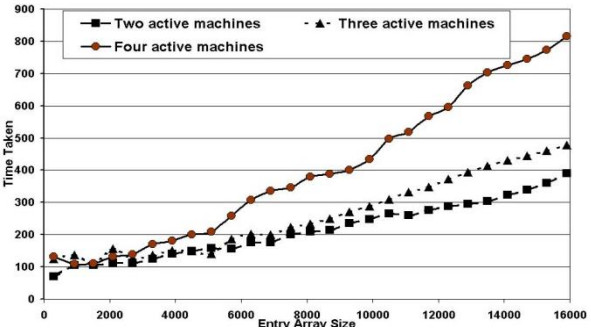


Fig 11: Comparing write operation of dynamic replica systems using a variant entry array size

Figure 11 shows the write operation performance comparison of two, three and four active spaces systems. This figure shows that; the performance of the dynamic replica write operation decreases by increasing the number of active spaces in the system. This is because the write 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.



. **Fig 12: Comparing write-read operation of dynamic replica systems using a variant entry array size**

Fig.12. Figure 12 shows the write-read operation comparison among two, three and four active spaces system using a variant entry size. From this figure we notice that; all curves are overlapped until the point 2700 bytes. Also, two and three active spaces dynamic replica curves are overlapped until the point 5100 bytes. This means that the difference in performance appears in the large entry array size. There are some test points in the two-active-space curve have greater value than the corresponding points in the three active space curve. This can be caused by the environment parameters such as network load or machine load.

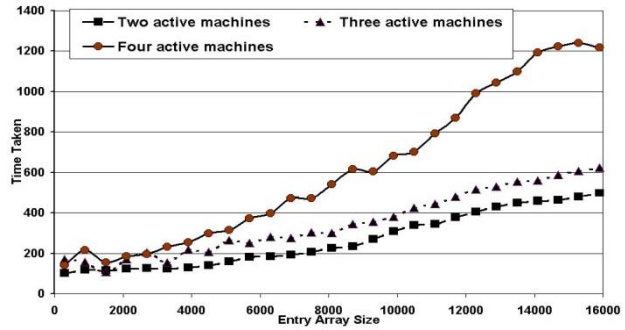


Fig 13: Comparing write-take operation of dynamic replica systems using a variant entry array size

Figure 13 shows the write-take operation performance comparison of two, three and four active spaces systems. From this figure, we can notice that; the four-active-spaces curve is the noisiest curve in the figure. The reason of this nose is that, by increasing number of machines (active spaces) the communication increases, so, the environment effect increases. Moreover, the difference between two and three-active-space curves is smaller than the difference between three and four active space curves.

6. SUMMARY

This paper introduces a new fault-tolerance mechanism in JavaSpaces (dynamic replica). The idea of this mechanism is to add a high availability layer to the JavaSpaces service. This mechanism guarantees that the user job can continue in spite of the tuple-space server failure. It discussed the dynamic replica system structure in details. Moreover, the complexity of dynamic replica operations has been introduced. Also, the practical tests of the dynamic replica systems are done. The failure test is done to insure that our system is a fault-tolerance system. The performance of the system is decreased by increasing the number of active spaces, but the system availability is increased. Moreover, the high availability layer test shows that the client in the dynamic replica system can continue its tasks in spite of the failure. In the future, we will extend this work on cloud systems to handle the failure of the uncertain resources.

7. REFERENCES

[1] Mattson T.G. Programming Environments for Parallel and Distributed computing: A Comparison of p4, PVM, Linda and TCGMSG. ftp Server, ftp.cs.yale.edu, 1995

[2] Milo Tomasevic Jelica Protic and Veljko Milutinovic. Distributed shared memory: Concepts and systems. IEEE Parallel and Distributed technology , 4(2):63-79, April, 1996

[3] Tran D., Nguyen T., and Motocova M., "Integrating Fault Tolerant Features Intoduction Topas Parallel Programmming Environment for Distributed Systems," International Conference on Parallel Computing in Electrical Engineering (PARELEC), Poland, pp. 453-459, 2002.

[4] Kai Hwang and Zhiwei Xu. Scalable parallel computing: technology, architecture, programming, volume 5 of Computer engineering series, pages 326–345. WCB/McGraw-Hill, 1st edition, February 1998.

[5] David Gelernter. Getting the job done (linda, parallel programming language). j-BYTE , 13(12):301–308, November 1988

[6] Fred B.Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. ACM Computing Surveys, 22(4):299-319, December 1990.

[7] M. Staroswiecki1, and A. Moradi Amani. Fault-tolerant control of distributed systems by information pattern reconfiguration. International Journal of Adaptive Control and Signal Processing JUN 2014.

[8] Oracle Technology Network. Jini Architecture Specification, available from Sun Microsystems WWW Site (http://www.jiniworld.com/doc/specs/html/devicearch-spec.htm), 2006.

[9] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Brigg, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-Tolerant Software Distributed Shared Memory. In the Proceedings of the 2015 USENIX Annual Technical Conference (USENIC ATC '15), 291-305, 2015.

[10] Mutasem Alsmadi, Usama A. Badawi and Hosam E. Refaat. High performance protocol for fault tolerant Distributed Shared memory(FaTP). Journal of Applied Science. 13(6) 790-799, 2013

[11] BlitzJavaSpacesImplementation.http://code.google.com/p/blitz-javaspaces/downloads/list, 2015

[12] Daniel Fiedler and Kristen Walcott and Thomas Richardson and Gregory M. Kapfhammer and Ahmed Amer and Panos K. Chrysanthis, Towards the Measurement of Tuple Space Performance, ACM SIGMETRICS Performance Evaluation Review, December, 2005, 33.

[13] van Heiningen, W., MacDonald, S. and Brecht, T. Babylon: middleware for distributed, parallel, and mobile Java applications. Concurrency Computat.: Pract. Exper., 20: 1195–1224,2008.

[14] Guanfeng Liang and Benjamin Sommer and Nitin H. Vaidya. Experimental performance comparison of Byzantine Fault-Tolerant protocols for data centers. INFOCOM'12", 1422-1430, 2012.