Journal of Computer Science 10 (4): 671-679, 2014
ISSN: 1549-3636
© 2014 Science Publications
doi:10.3844/jcssp.2014.671.679 Published Online 10 (4) 2014 (http://www.thescipub.com/jcs.toc)

SERVER FAILURES ENABLED JAVASPACES SERVICE

[1]Mutasem K. Alsmadi, [2]Usama A. Badawi and [2]Hatem M. Moharram

[1]Department of MIS, Collage of Applied Studies and Community Service, University of Dammam, Saudi Arabia
[2]Department of Mathematics, Computational Science Division, Faculty of Science, Cairo University, Egypt

Received 2013-10-28; Revised 2013-11-23; Accepted 2013-12-21

ABSTRACT

JavaSpaces service is a Distributed Shared Memory (DSM) implementation. It has been introduced by Sun Microsystems as a service of the Jini system. Currently, JavaSpaces support client side fault tolerance. It enables both transaction and mobile coordination mechanisms for such purpose. The application failures could be detected and recovered. However, server side failures may occur during the application runtime. Therefore, it is important to supply JavaSpaces with a mechanism that handles such type of failures dynamically. On the other hand, An example of a system that supports both server and client fault tolerance over DSM is TRIPS system. TRIPS protocols are suitable to be integrated in JavaSpaces to supply it with server fault tolerance capabilities. In this study, a server Failures Enabled Javaspaces Service (FTJS) is introduced. FTJS is based on the dynamic failure detection and recovery mechanisms implemented by TRIPS. However, FTJS is able to handle both client and server side failures. The analysis, design and implementation issues of FTJS are introduced.

Keywords: Distributed Application, Dynamic Recovery, Failure, JavaSpaces

1. INTRODUCTION

Machine crashes and network partintions are major problems while running a distributed application. It is important to deal with failures that are caused by such events within runtime. Otherwise, it will be a must to restart the application from the beginning. A possible solution to this problem is to introduce a software layer that is able to detect failures and recover from them dynamically. Fault tolerance mechanisms, such as transactions and mobile coordination, are applicable to deal with client failures. Other mechanisms, such as dynamic replication, are suitable to the server failures.

Sun Microsystems has introduced the Jini system. It is a distributed system that enables groups of users and the resources required by those users to be federated. The main goal of Jini is to facilitate different resources to be available for cleints over the network. Moreover, JavaSpaces is a service introduced by the Jini system. It is a Distributed Shared Memory (DSM) used for object storage and communication (SM, 2007; Kanjilal, 2013). JavaSpaces service has been supplied with transactions and mobile coordination mechanisms. Therefore, it is able to deal with the client side failures. It is important to support JavaSpaces service with server failures handling mechanisms as well (Kamalam and Bhaskaran, 2012).

On the other hand, TRIPS is a system that enables dynamic detection and recovery of failures in both the client and server sides using the dynamic replication over DSM (Badawi, 2009).

1.1. Problem Statement

The goal of this research work is to construct FTJS, which is a server failure enabled JavaSpaces service. This will be accomplished by integrating the dynamic failure detection and recovery mechanisms introduced by TRIPS in the JavaSpaces service. This will enable JavaSpaces to deal with both client and server failures.

1.2. Related Studies

TRIPS enables DSM based applications to tolerate with both server and client failures. It is based on the Linda Model and constructs a distributed environment

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

for parallel processing. The Tuple space concept has been introduced by the Linda Model. Tuple space could be defined as an associative Shared Memory (DSM) accessible to all application processes. Its contents are entries, which are retrieved using a matching mechanism by their contents rather than by physical addresses (Badawi, 2009; Alsmadi *et al*., 2013). A DMS access set of operations has been introduced by Tuple Space.

1.3. TRIPS System Structure

TRIPS is structured in three main layers as shown in **Fig. 1**, namely, the transis layer, LiPS Layer and Trips message handling layer. The Transis event layer is a group communication layer inherited from the Transis group communication system. It is focused towards high throughput local communication. It supports group communication service. Transaction based delivery semantics are guaranteed. Message ordering is supported and network failures are transparentfrom the user. If membership changes occure, the system reports them. The idea behind its mechanism is to create a singlton group for each newly arriving process. The new group receives a ‘mailbox’ to which messages arrive (Dolev and Malki, 1996; Liefke, 1998). The Transis Event Layer is composed of two sub-layers, namely, the network layer and the group communication layer. The former layer is responsible for handling socket connection and physical data routing. The group communication Layer facilitates the membership mechanisms that enable group members to identify the group communication and configuration mechanisms that enable the member to communicate and broadcast messages to the other members (Badawi, 2009).

The second TRIPS layer is the LiPS-layer. This layer controls and manage the distributed applications. This is accomplished through control processes called lipsds. Lipsds are responsible for managing the DSM and application message log. They start and control the application processes. Moreover, they replicate the application processes data to other equivalent processes. Server level failures are handled using replication. This layer is composed of two sub-layers, namely, Trips middle layer and local tuple space layer. The former includes the interface operations enabling the application to interact with the DSM. Examples are Mid_in(), to extract entries, Mid_out(), to write entries and Mid_rd(), to read entries. LiPS system (Library of Parallel Systems) implementation of Linda premitives is used in constructing these operations. The local tuple-space layer includes the DSM structures. This layer is used as the system repository and is inherited from the LiPS tuple space structure (Setz, 1997).

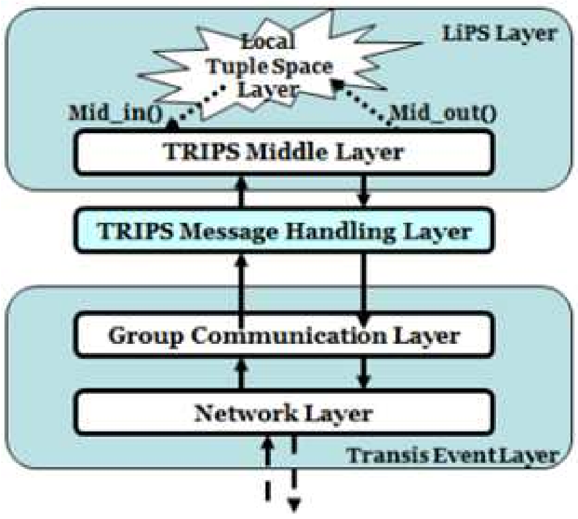

Fig. 1. TRIPS system internal layers

TRIPS message handling layer is responsible for dealing with different message types. The fault tolerant mechanism that handles different message types is integrated in this layer. The main component in this layer is the “State Change Protocol”, that handles both regular distributed shared memory messages and configuration change ones. This protocol is activated as soon as a message is received either from a member to access the DSM, or from the membership layer indicating view change.

1.4. The JavaSpaces Service

JavaSpaces has an associative set of operations to access the contents of the space. This set of operations has inherited its behavior from the Linda tuple space model. For example, to insert an entry to the JavaSpaces the write() operation is used. To extract an entry fro the JavaSpaces the take() operation is used. The write() and take() operations are equivalent to the Linda operations out() and in() respectively (Busi *et al*., 2010).

JavaSpaces service supports transactions and mobile co-ordination to enable client side failures handling. Transaction methodology enables all operations to be performed under it. For example, if a take() operation is done under a transaction, the entry is added to a set of entries that are taken by the transaction. If the transaction is aborted, the taken entries are returned to the space. The taken entries are removed from the space after the transaction is committed (SM, 2007). On the other hand, mobile co-ordination is more associated with the DSM concepts. In this method the coordination primitives (JavaSpaces operations) are moved to the server side, which contains the space that the client wishes to access.

JavaSpaces operations that are executed under mobile co-ordination must be encapsulated into a coordination method. This method is executed by the JavaSpaces server (Rowstron, 1999; Lazr, 2001; Tanha *et al*., 2012).

JavaSpaces has been introduced as one of the Jini system powerful services. Jini system is introduced by Sun Microsystems. JavaSpaces enables the Java environment to deal with a network of virtual machines. It helps in constructing variant sized distributed applications. The central element in Jini is the service, which is an interface of hardware device, application, database, or anything that can be connected to the network. To enable a device with Jini technology, it must have a processing power and memory. Jini enables devices without memory or processing power to be connected to the vertual system and controlled by other hardware and/or software, proxies. Such proxies task is to present the device to the system with processing power and memory (Heiningen *et al*., 2006a; 2006b).

JavaSpaces is a DSM implementation. It stores data items, called entries, to be accessed by clients. The entry objects are expressed in classes that implement the interface Jini.core.entry.Entry. Entry behavior and characteristics are inherited from the Linda tuple space model. Different entries are said to be of the same type if they are members in the same class. The entry can have methods that define its behavior (SM, 2007; Batheja and Parashar, 2010; Marghny and Refaat, 2012). In this section, the Jini system structure is viewed as well as the current JavaSpaces fault tolerance protocols.

2. MATERIALS AND METHODS

The methodology in this research work is based on the idea of integrating the TRIPS systems, that enables server failures in the JavaSpaces service that enables applications failures within runtime. In this section, the TRIPS fault tolerance methodology and the Jini system structure are introduced.

2.1. Fault Tolerance in TRIPS

Dynamic replication is the mechanism used by TRIPS to enforce fault-tolerance. The core of the TRIPS message handling layer is the scheduler that is responsible for receiving and recognizing the type of the state change message. Then it is responsible for directing the message to the suitable handling routine (Badawi, 2009). The scheduler structure and vehavior is whown in **Fig. 2**. In case of configuration change during handling a regular DSM message, an interrupt request is sent to the DSM handler. The DSM operation is intruppted and the control is returned to the scheduler without performing the DSM operation. The configuration changes are handled first. The Canceled operation is inserted in a local queue to be accessed later.

TRIPS uses the “State Change Protocol” to ensure the availability of the distributed application processes. This protocol is responsible for handling the possible state changes, such as new member join or existing member exit, that could occure to the distributed application. The protocol guarantees the survival of data in the DSM in spite of failures. Moreover, it makes sure that the regular operations are applied to all members in the configuration. In case of starting a new member, the global queue that contains all application members is activated and the DSM data structures are initialized. Then, control is passed to the configuration change handler that controls the membership changes (Badawi, 2009).

2.2. The Jini System Structure

To accomplish the service communication, Jini uses Remote Method Invocation (RMI) as shown in **Fig. 3**. RMI enables full objects (code and date) to be passes around the network. This gives Jini the simplicity of moving encapsulated objects around network. From the figure, one can notice that Jini layers are located on top of the Java platform. This enables the processes and services, that run under Jini control to inherit the powerful behavior of java processes. Jini network federation consist of two main layers. The Lookup layer includes a protocol that enables clients to search for the Jini services they need to utilize. The Discovery/Join layer includes discovery and join protocols that enable the clients to join the services they need to utilize.

3. TRIPS JAVASPACES SERVICE (FTJS)

Both of the JavaSpaces fault tolerance methods are dealing with client side failures. The proposed service (FTJS) deals with both server and client side failures. For this purpose, a warm backup replication protocol is presented. In this section, the proposed protocol, SpacesManager, is introduced as well as the analysis and design of FTJS.

3.1. The SpacesManager Layer

The idea behind FTJS is to construct the SpacesManager layer that increases the system availability. Normally, there exist many running JavaSpaces services 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. The SpacesManager layer is responsible for spreading the effect of the client operations in all active spaces. If the client writes an entry in the system, the SpacesManager replicates this entry in all active spaces and ensures that all spaces are identical.

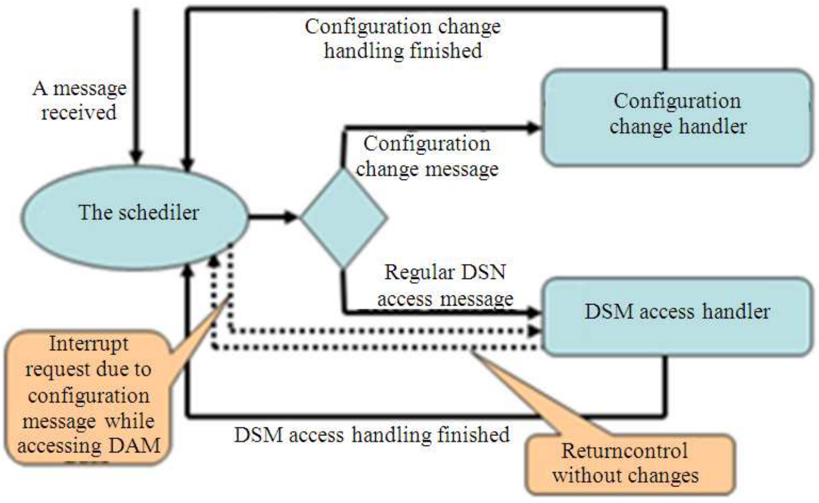
A message received
Configuration change handling finished
Configuration change handler
Configuration change message
The schediler
Regular DSN access message
DSM access handler
Interrupt request due to configuration message while accessing DAM
DSM access handling finished
Returncontrol without changes

Fig. 2. TRIPS System Main Function (The Scheduler)

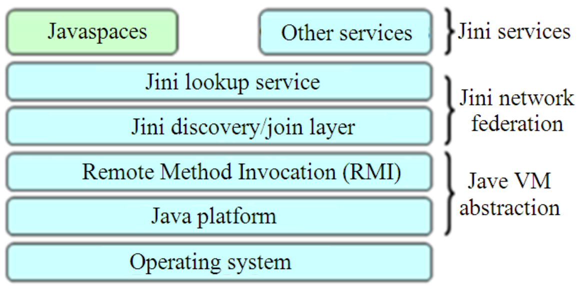

Fig. 3. The jini system structure

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 SpacesManager failure recovery algorithm is shown in **Fig. 4**.

The SpacesManager layer 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 the replica or not. If the failed machine is the replica, one of still alive active spaces is chosen to be the original space. To survive an active space 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. If any of the active spaces other than the replica 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, the SpacesManager blocks this machine. In other words, the system will delete this machine from the active spaces list. If the failed/disjoined machine comes back to the system, the SpacesManager deletes all entries in its JavaSpaces and rejoins it as a passive machine.

```
Spaces manager failure recovery algorithm
initialization :
check the excisting 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 javaspace;
If machine failure occur 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 choosen to be the original
         spaces;
one of the passive spaces is choosen to be the acive;
clear all enteries in the new active space;
copy all enteries 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 choosen to be the acive;
clear all enteries in the new active space;
copy all enteries from any of still alive active spaceto the
         new active space ;
if if the failed space is passive then
block this machine;
if one of the failed machine comes back to life then
delete all enteries in its space;
join the machine as passive machine;
 unblock this machine;
until no condition;
```

Fig. 4. SpacesManager recovery algorithm

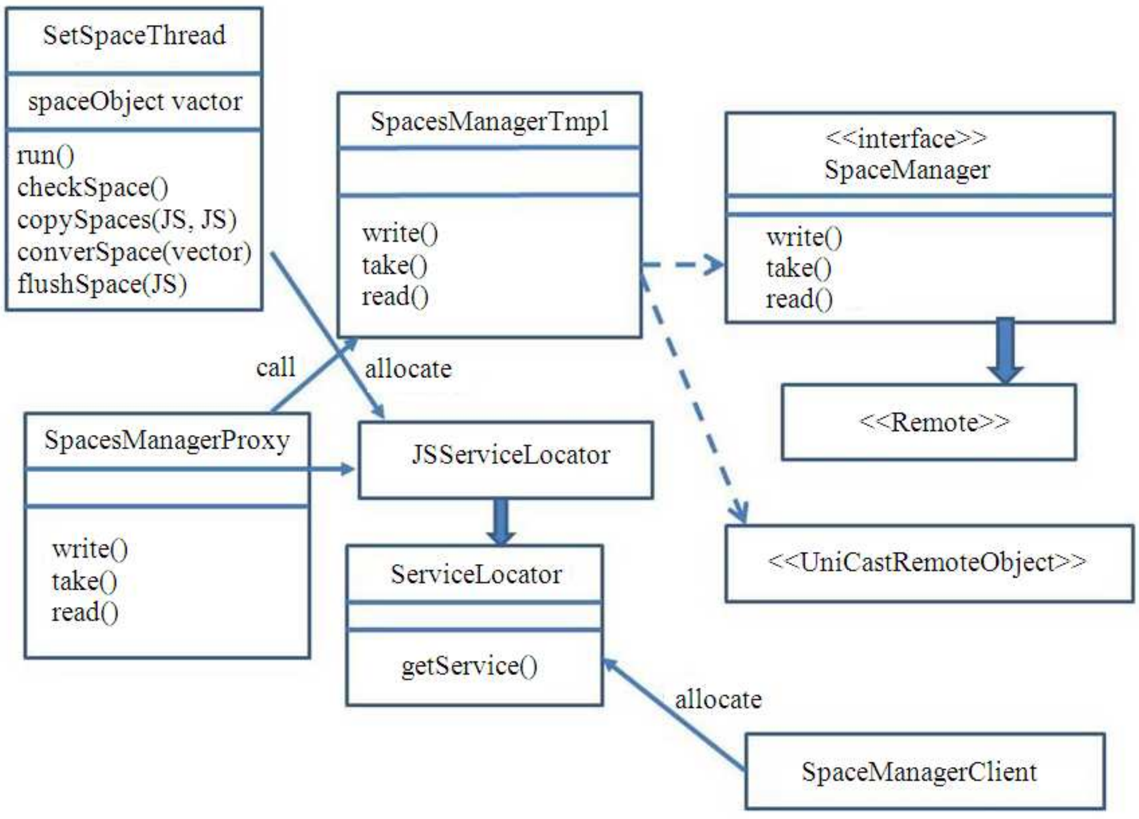

Fig. 5. FTJS Main Components (Class diagram)

3.2. FTJS Service Design

FTJS service consists of three main parts. **Figure 5** shows the FTJS class diagram. The first part is the SpacesManager. It is based on defining a DSM control service in the Java RMI. The SpacesManager interface contains the basic DSM operations (write(), take() and read()). This interface extends the Java API Remote interface. SpacesManagerImp is a class that implements the SpacesManager interface and extends the java API interface Unicast-Remote-Object. 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. The GetSpacesThread class contains a public variable of type vector called SpacesObject. It contains objects of all JavaSpaces services in the system and other metadata like the type of space (active or passive), block...

The second component of FTJS is the SetSpacesThread that is responsible for managing the failures. It uses the checkSpaces() method to check the existence of the system machines. This method accesses, in turn, the JSServiceLocator class objects 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. **Figure 6** shows the FTJS structure.

The third component of FTJS service is the SpacesManagerClient, which is a client program that is used to test the service using the resizable entry MyEntry. The client program fetches the dynamic replica service using the ServiceLocator class. 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. **Figure 6** gives an overview to the flow control in the dynamic replica protocol used in the FTJS service.

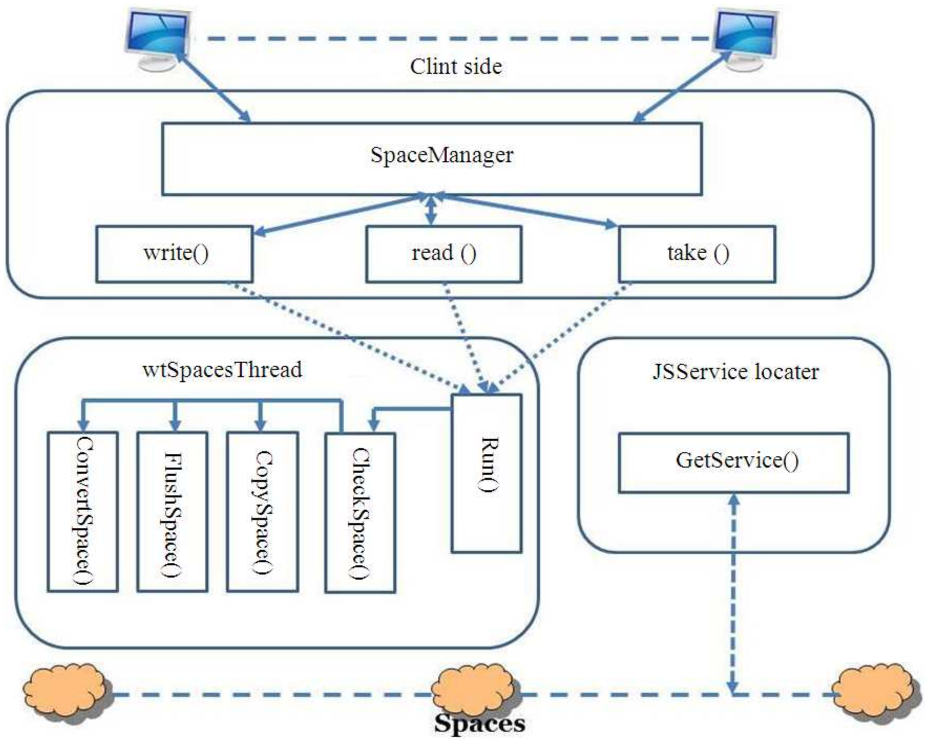

Fig. 6. FTJS internal structure

4. RESULTS

In this section, practical tests are introduced to evaluate the FTJS service. First, the test environment and technique are introduced. Then the tests and their results are presented.

4.1. Test Environment and Technique

The measurements are performed by using six PC's each with a CPU of type Intel Pentium 2.4 G.H and 512 RAM. The inter-communication 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, Jini(TM) Technology Starter Kit v2.0.2 and a free visual platform for Jini 2.0 that is called Inca X(TM).

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 service by testing the DSM access operations for insertion and retrieval.

4.2. The Fault Tolerance Test

In this section, it is proved that the proposed service tolerates with failures. The following scenario has been applied for this purpose. A counter is intiated by one client. 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 above steps are repeated in a large number of iterations. One of the active spaces is enforced to fail during the process. If the client process survives in spite of the failure and the counter increases correctly, then it is proved that the service is fault tolerant.

Figure 7 shows a skeleton code for the test steps. In this test, the 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 writing and taking entry. The second part of the list (B) shows the setSpacesThread 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 JavaSpaces is enforced to fail. The FTJS 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, FTJS service converts the passive spaces1 to active spaces1 and blocks the object of passive spaces1 (not exist).

```
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

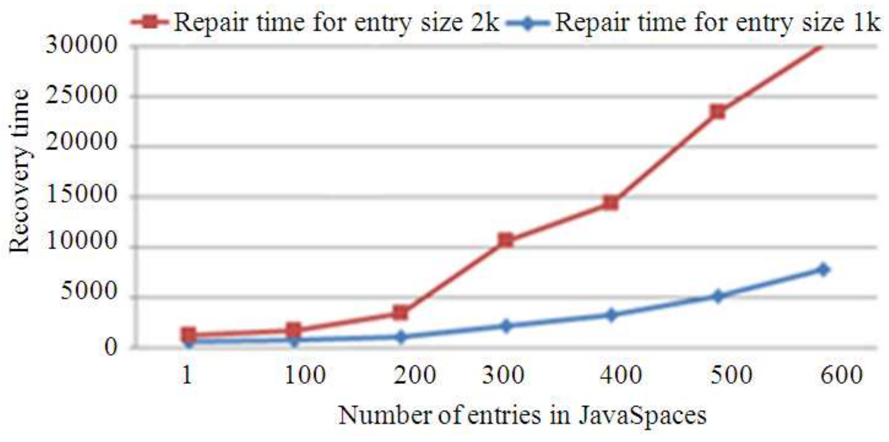

Fig. 9. Recovery time in FTJS

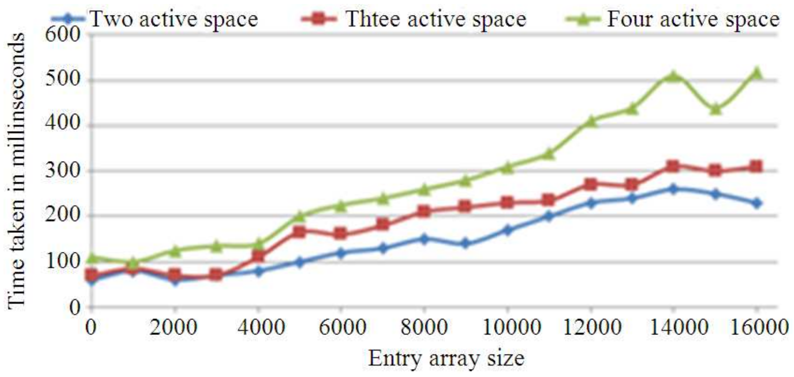

Fig. 10. System performance comparison for 2, 3 and 4 active spaces (Write operation)

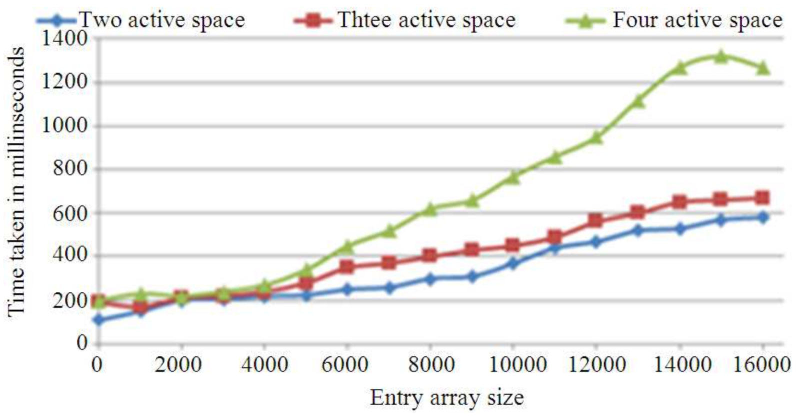

Fig. 11. System Performance Comparison for 2, 3 and 4 active Spaces (Write-take operation)

4.3. Measuring the Recovery Time

FTJS recovery time 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** illustrates the recovery time in FTJS. From the figure, it is clear that increasing the number of entries in the space leads to increasing the recovery time. This is due to the time taken to copy the entries to the new active space.

4.4. Performance Tests

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

Figure 10 shows the write() operation performance in the cases of two, three and four active spaces. The figure shows that the performance of the 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 minimal at the small entry array size.

Figure 11 shows the write()-take() operation performance comparison for two, three and four active spaces. From this 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 extra communication time. Moreover, the difference between two and three-active-space curves is smaller than the difference between three and four active space curves.

5. CONCLUSION

In this research work, the FTJS service is introduced. It is a server failures enabled JavaSpaces service. A high availability layer called SpacesManager layer has been added to the JavaSpaces service. If a failure occurs, the application data survives without any interruptions.

Moreover, the detection and recovery process is transparent to the user service.

Many types of practical tests have been applied to show the proposed service performance. A fault tolerance test has been performed as well as a recovery time test, performance tests have been appled on different read-write premitives. All the tests have proved that the service performance is reasonable. It is also shown that the proposed service is practically applicable. The proposed JavaSpaces service has been applied To Local Area Network (LAN). It is possible to apply it in the Wide Area Network (WAN) in later versions. On the other hand, the current version of the service cannot deal with the case of merging spaces with entries inside. The non-original space must be empty in case of merge. A possible future work is to enhance the protocol to deal with non-empty spaces merge. This requires a lot of work to deal with the famous merging conflicts.

6. REFERENCES

Alsmadi, M., B.A. Usama and S. Reffat, 2013. A high performance protocol for fault tolerant distributed shared memory (FaTP). J. Applied Sci., 13: 790-799.

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

Batheja, J. and M. Parashar, 2010. A framework for opportunistic cluster computing using javaspaces. Proceedings of the 9th International Conference on High-Performance Computing and Networking, Jun. 25-27, Springer-Verlag, London, pp: 647-656.

Busi, N., R. Gorrieri and G. Zavattaro, 2010. Process calculi for coordination: From linda to javaspaces. Proceedings of the 8th International Conference, Algebraic Methodology and Software Technology, May 20-27, Springer-Verlag London, pp: 198-212. DOI: 10.1007/3-540-45499-3_16

Dolev, D. and D. Malki, 1996. The Transis approach to high availability cluster communication. Commun. ACM, 39: 64-70. DOI: 10.1145/227210.227227

Heiningen, V.W., T. Brecht and S. MacDonald, 2006a. Babylon v2.0: Middleware for distributed, parallel and mobile Java applications. Proceedings of the 20th International Parallel and Distributed Processing Symposium, Apr. 25-29, IEEE Xplore Press, Rhodes Island. DOI: 10.1109/IPDPS.2006.1639498

Heiningen, V.W., T. Brecht and S. MacDonald, 2006b. Exploiting dynamic proxies in middleware for distributed, parallel and mobile java applications. Proceedings of the 20th International Parallel and Distributed Processing Symposium, Apr. 25-29, IEEE Xplore Press, Rhodes Island. DOI: 10.1109/IPDPS.2006.1639504

Kamalam, G.K. and V.M. Bhaskaran, 2012. Novel adaptive job scheduling algorithm on heterogeneous grid resources. Am. J. Applied Sci., 9: 1294-1299. DOI: 10.3844/ajassp.2012.1294.1299

Kanjilal, J., 2013. Understanding the JINI Networking Technology. ASPAlliance.

Lazr, I., 2001. Designing a fault-tolerant JINI compute server. Pennsylvania State University.

Liefke, T., 1998. Extension of the trips prototype. Technical Report: Darmstadt University, Department of Theoretical Computer Science.

Marghny, M.H. and H.E. Refaat, 2012. A new parallel association rule mining algorithm on distributed shared memory system. Int. J. Bus. Intell. Datamin.

Rowstron, A.I.T., 1999. Mobile co-ordination: Providing fault tolerance in tuple space based co-ordination language. Proceedings of the 3rd International Conference on Coordination Languages and Models, (LM ‘99), Springer-Verlag London, pp: 196-210.

Setz, T., 1997. Software Fault-Tolerant Distributed Applications in Lips. 1st Edn., Univeristat des Saarlandes, Saarbrücken, pp: 20.

SM, 2007. JavaSpaces Specification. Sun Microsystems Found.

Tanha, M., S.D.S. Torshizi and S. Shamala, 2012. A discrete event simulator for extensive defense mechanism for denial of service attacks analysis. Am. J. Applied Sci., 9: 909-916. DOI: 10.3844/ajassp.2012.909.916