TS-PVM: A Fault Tolerant PVM Extension for Real Time Applications

Usama Badawi
Department of Mathematics, Faculty of Science, Egypt

Abstract: *In this research work, a fault tolerant extension of the de facto message passing system parallel virtual machine, TS- parallel virtual machine, is introduced. This extension enables real time applications over parallel virtual machine. In PVM and similar message passing systems, if the message receiver is not available, due to network failure or machine crash, a failure is reported and data must be resent. If the transferred data is persistent, i.e. should not be lost, then it is important to provide the system with failure recovery mechanisms. The idea behind the proposed extension, TS- parallel virtual machine, is to integrate a fault tolerant distributed shared memory layer, in parallel virtual machine. Such layer has been inherited from the TRIPS system that supports fault tolerance over a DSM. The challenge of this research work is the integration of a DSM layer in a message passing system. The proposed parallel virtual machine extension would have bad performance if compared to the pure message passing one. On the other hand, the real time application can be completed in spite of failures occurrence.*

Keywords*:* *Fault tolerance, persistent data, PVM, real time applications, tuple space.*

Received January 14, 2007; accepted May 12, 2008

1. Introduction

Parallel Virtual Machine (PVM) is a standard message passing system. It has gained its standardization because of its simple set of operators that enables users to parallelize the program by adding few lines to the existing sequential code. It is designed to link computing resources and to provide users with parallel platforms to run their applications [3]. In spite of the many advantages of the PVM system, it still lacks for being fault tolerance. Such failures may lead not only to cause performance problems but also to stop the application execution. Sometimes, the transferred data in real time applications, such as mobile communication, is Persistent Data (PD) i.e., should not be lost. For this type of applications, failure recovery must be guaranteed. Integrating a fault tolerance layer that introduces dynamic detection and recovery mechanisms to the system is one solution for this problem. PVM has been supplied by failure detection mechanisms that enable the system to identify some types of failures such as machine loose. It has no way to recover from the failure dynamically [13].

On the other hand, TRIPS is a fault tolerance system that introduces a DSM that enables dynamic failure detection and recovery using dynamic replication [8]. In this research work, a proposed fault tolerant extension of PVM, called TS-PVM, is introduced. The proposed extension is based on the idea of integrating the DSM fault tolerance layer introduced by TRIPS in PVM. TS-PVM is the suitable solution for real time applications with persistent data PDRTA such as mobile communication.

2. Related Works

One example of systems that support PVM with fault tolerance capabilities is the Task-Oriented Parallel progrAmming System environment (TOPAS). One of the system objectives is to provide the parallel application with facilities for fault tolerance [12]. TOPAS uses transactions to ensure all or none delivery of messages among different clients. TOPAS automatically analyzes data dependence among tasks and synchronizes data, which reduces the time needed for parallel program developments. It also provides supports for scheduling, dynamic load balancing and fault tolerance. Experiments show simplicity and efficiency of parallel programming in TOPAS environment with fault-tolerant integration, which provides graceful performance degradation and quick reconfiguration time for application recovery.

Another example is the new general purpose transport protocol called Stream Control Transmission Protocol (SCTP) [10]. SCTP associations support multi-homing to provide redundancy at the endpoint level, which increases connection level fault tolerance. The characteristics of SCTP closely match with the message passing semantic of PVM and have motivated the development of SCTP-PVM, a PVM extension, enabled to take advantage of the features of SCTP for direct communications among PVM tasks. In SCTP-PVM, the messages among tasks are directly mapped in SCTP associations and streams. Messages that must reach the same task with different tags can be assigned to different streams and can travel in the network independently. In addition, the PVM standard library is extended to permit programmers to use the SCTP protocol [13].

3. The TRIPS System

TRIPS is a system that enables DSM based applications to tolerate with failures. It constructs a distributed environment for parallel processing using the Linda model. Linda model has presented the tuple space concept, which is an associative shared memory DSM accessible to all application processes. It is associative in the since that it contains entries, which may be retrieved by their contents rather than by physical addresses, using a matching mechanism [1]. It introduces a set of operations to access the DSM. This section introduces TRIPS system structure and its fault tolerance mechanism.

3.1. TRIPS System Structure

TRIPS consists of three main layers as shown in Figure 1. The first layer is the transis event layer. It is a layer inherited from the transis group communication system. This layer is geared towards high throughput local communication. Transis-layer supports group communication service with strong semantics. All-or-none delivery semantics are guaranteed. Transient network failures are transparent, and message ordering is supplied. It reports membership changes. Once a process joins the application, it creates a singleton group, and receives a "mailbox" to which messages arrive [2], [4]. This layer is composed of two sub-layers, namely, the network layer and the communication layer. The network layer is responsible for handling the socket connection and the physical routing of the data. The communication layer includes the membership mechanisms that enable the member to identify the group configuration, and communication mechanisms that enable the member to communicate and broadcast messages to the other members [1].

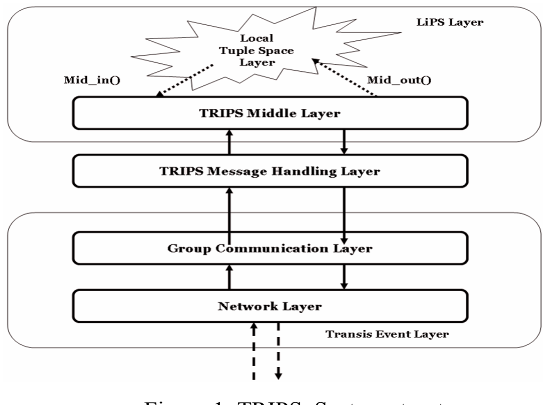


Figure 1. TRIPS System structure.

The second layer in TRIPS is the Library of Parallel Systems (LiPS) Library of Parallel Systems). This layer introduces control processes for distributed applications, called lipsds. They manage the DSM and application message log, start and control the application processes, and replicate their data to other equivalent processes. Server failures are handled using replication. This layer is composed of two sub-layers, namely, Trips middle layer and local tuple space layer. TRIPS middle layer includes the interface operations that enable the application to interact with the DSM. Examples are Mid_out(), to write entries, Mid_rd(), to read entries, and Mid_in(), to extract entries. These operations are constructed using the Linda tuple space model implemented by the LiPS system. The local tuple-space layer includes the DSM structures. The LiPS tuple space structure is used to construct this layer as a system repository. All processes are controlled by the lipsd located on their machine [9].

The third layer in TRIPS is the TRIPS message handling layer that includes the fault tolerance protocol, which handles different types of messages. It includes a protocol, called "state change protocol", to handle both configuration change and regular DSM messages. This layer is activated as soon as a message is received either from one of the members to access the DSM, or from the membership layer indicating view changes.

3.2. Fault Tolerance in TRIPS

TRIPS uses dynamic replication to enforce fault-tolerance. An important part of the TRIPS message handling layer is the scheduler that is responsible for receiving the state change message and deciding its type, i.e., either configuration (membership) change message or a regular DSM access message. Then it directs the message to the suitable routine to be handled [1]. Figure 2 shows the scheduler and its behavior. If a configuration change message arrives to the scheduler while handling a DSM message, an interrupt request is sent to the DSM handler. Control is returned to the scheduler without performing the DSM operation to handle configuration changes first.

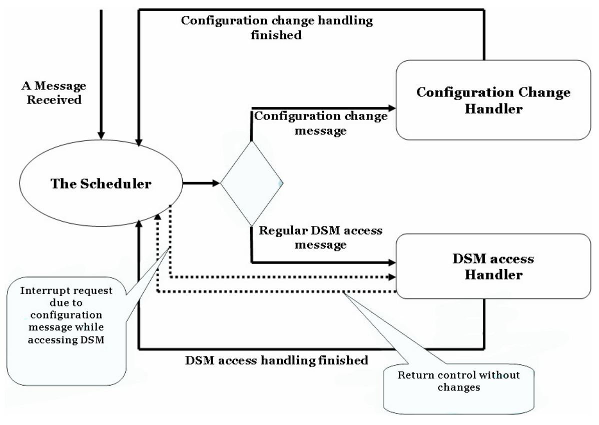

Configuration change handling finished
A Message Received
Configuration Change Handler
Configuration change message
The Scheduler
Regular DSM access message
DSM access Handler
Interrupt request due to configuration message while accessing DSM
DSM access handling finished
Return control without changes

Figure 2. TRIPS scheduler.

To ensure the availability of the distributed application processes, TRIPS uses the "State Change

Protocol". This protocol is responsible for handling the possible state changes occurring to the distributed application. The protocol guarantees that the data is not lost in the DSM in spite of failures. Whenever a new member is started, the global queue and the DSM data structures are initialized. Then, the control is passed to the configuration change handler, which is responsible for membership change events handling. State changes are categorized into two main types; regular DSM changes and membership configuration changes [1].

4. The Proposed Extension

PVM is a message passing system that permits a heterogeneous collection of computers networked together to be viewed by a user's program as a single parallel computer [11]. It is designed to link computing resources and to provide users with parallel platform to run their computer applications, irrespective of the number of computers they use and where the computers are located. In this section, the characteristics and structure of the PVM system are presented. Then the modifications required to build the proposed PVM extension, TS-PVM, are introduced as well.

4.1. PVM Structure and Behavior

PVM transparently handles all message routing, data conversion, and task scheduling across a network of heterogeneous architectures [5]. Messages are tagged at sending time with a user defined integer code and can be selected for receipt by source address or tag. The sender of a message does not wait for an acknowledgment from the receiver, instead, it continues as soon as the message has been handed to the network and the message buffer can be safely deleted or reused. If one node sends several messages to another, they will be received in the same order. PVM is also able to withstand host and network failures. It does not recover an application after a crash, it only provides polling and notification primitives to allow fault tolerant applications to be built. The virtual machine is dynamically reconfigurable [7]. Its tasks may possess arbitrary control structures. At any execution point of concurrent applications, any task may start/stop other tasks or it may add machines to or delete ones from the virtual machine. Any process may communicate and synchronize with other processes. The PVM system structure is based on two main components, namely, the PVM daemon, called pvmd, and the system function library of PVM routines. These two main components are discussed here.

The pvmd is a control process that is responsible for controlling the behavior of the application processes running on a given host participating in the virtual machine. In order to reduce security risk and minimize the impact of one PVM user to another, pvmds owned by one user do not interact with those owned by others. The pvmd serves as a message router and controller. It provides a point of contact, authentication, process control, and fault detection. A pvmd occasionally checks that its peers are still running. Even if application programs crash, pvmds continue to run, to aid in debugging [6].The first pvmd is designated as the master one, while the others; started by the master; are called slaves. During normal operation, all are considered equal. Only the master can start new slaves and add them to the configuration. Figure 3 shows PVM processes communication. The communication between any two processes on top of PVM must be done through the pvmds allocated on their machines.

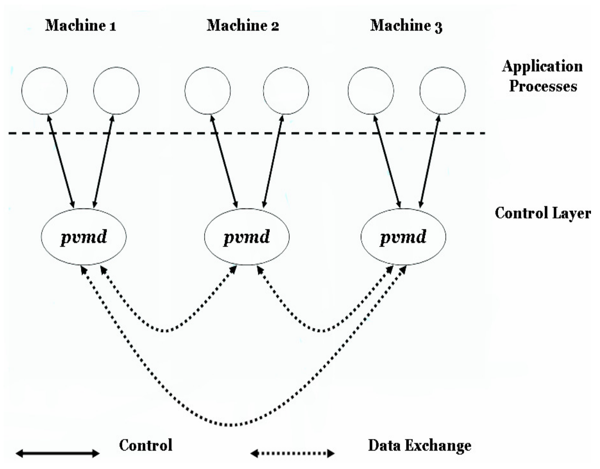


Figure 3. Process communication in PVM.

The PVM function library, libpvm allows a task to interface with the pvmds and other tasks. It contains functions for packing and unpacking messages, and functions to perform PVM system calls. The top level of the libpvm library includes the programming interface functions such as *pvm-send()* to send a message and *pvm-receive()* to receive a message. For simplicity, the proposed extension will be applied to these two functions. The protocol can be easily generalized to other interface functions. These functions are written in a machine independent style.

The bottom level is kept separate and can be modified or replaced with machine specific files when porting PVM to a new environment. Message packing in the *pvmd* is very simple. Messages are handled using a structure called mesg. There are encoders for different data types to be utilized within PVM modules.

Pvmds communicate through UDP sockets. Since the UDP sockets introduce an unreliable delivery, therefore packets can be lost, duplicated or reordered.

An acknowledgment and retry mechanism is used. UDP also limits packet length; therefore, PVM fragments long messages. Pvmds usually do not communicate with foreign tasks (those of other users). The pvmd has message re-assembly buffers for each task it manages. To free up the re-assembly buffer for a foreign task, if the task dies, the pvmd would have to request notification from the task's pvmd. A PVM task can send messages directly to another task via one of the point-to-point communication primitives, or across a set of tasks by first joining the corresponding group and then using the appropriate communication primitive. However, PVM groups can be dynamically created [6].

4.2. TS-PVM Structure

PVM does not guarantee PD delivery. Therefore, it is not suitable for real time applications with persistent data. It should be provided with fault tolerance mechanisms in order to survive the application. The TS-PVM extension handles the application processes and daemons on different machines as TRIPS application processes. The PVM user should not notice any change in his interface primitives. Therefore, the PVM send and receive primitives, *pvm-send()* and *pvm-receive()*, are used in the proposed extension. In the lower level, *pvm-send()* and *pvm-receive()* are used as masks for the TRIPS corresponding primitives, *Mid_out()* and *Mid_in()* respectively. To construct the proposed protocol, two layers, namely the interface layer and the kernel layer, have been integrated in PVM.

4.2.1. TS-PVM Interface Layer

The interface layer deals with PVM user interface functions. PVM has a compact set of interface functions that enable the user to interact with the low-level system routines. In the interface layer, the PVM standard set of interface operations are used as masks for the TRIPS primitives that are used actually to access the DSM. Each PVM primitive contains a call to the corresponding TRIPS primitive that handles the actual work. The attributes of the PVM primitive are directed to be allocated as parameters to the corresponding TRIPS primitive.

Figure 4 shows the TS-PVM interface layer. The Interface layer routines use the data in the *pvm-send()* call to generate a new entry to the DSM using *Mid_out()*. Similarly, the data extracted from the DSM using the *Mid_in()* call is used to be facilitated by the *pvm-receive()* call. A new entry is generated for each message sent through the virtual machine that includes data about the sender, the receiver, and the message body. All this information is accessed using the attributes of the PVM interface functions.

4.2.2. TS-PVM Kernel Layer

This layer includes routines to enable lipsd; the TRIPS control process; to control the behavior of the pvmd located on the same machine. pvmds must be under the control of the TRIPS daemons (lipsds), therefore, the communications among the pvmds do not occur in a direct way as usual. Instead, every pvmd sends its requests to the lipsd located on the same machine. In this case, all operations done by the PVM real time application processes are controlled by the TRIPS dynamic allocation routines. To construct this layer, many integrity constraints are applied. One important constraint is to make the CLUSTER file, which defines the set of hosts participating in the TRIPS configuration and the HOSTFILE, which defines the set of machines participating in the PVM virtual machine, have the same members. Otherwise a conflict may occur while handling the send/receive requests.

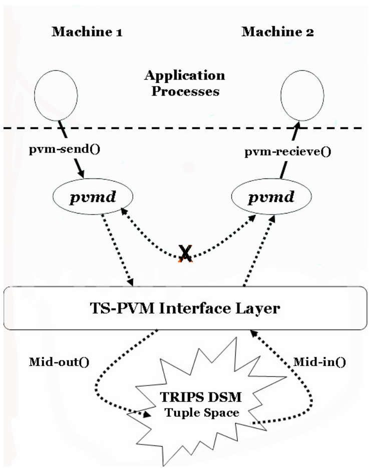


Figure 4. TS-PVM Interface layer.

To run the applications under this protocol, the following steps are applied. First, TRIPS system is started, then the master pvmd is started. It is started under the control of TRIPS runtime. There exists a lipsd on the same machine. Then, the slave pvmds are started in the regular way. All these steps are considered preparation steps before starting the PVM application. Figure 5 shows the following facts. First, pvmd-pvmd communication is performed through the TRIPS DSM. Direct communication among pvmds is not allowed.

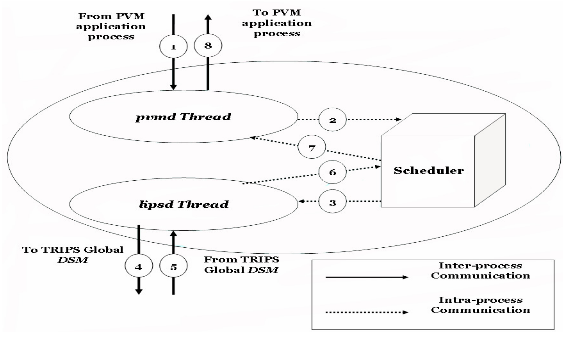

From PVM application process
To PVM application process
1
8
pvmd Thread
2
7
6
3
Scheduler
lipsd Thread
To TRIPS Global DSM
4
5
From TRIPS Global DSM
Inter-process Communication
Intra-process Communication

Figure 5. TS-PVM Kernel layer.

Any PVM interface request is converted, by the TS-PVM interface layer, and is directed to the local TRIPS DSM. Moreover, all application processes and pvmds are under the control of the lipsd located on the same machine. This enables lipsds to be the main controller on each machine. Moreover, lipsds are linked to a separated DSM, different from the TS-PVM local DSM. This is done to trace the existence of different lipsds and to enable the data exchange among different lipsds.

4.2.3. TS-PVM Control Processes

On each machine participating in the virtual machine, there exist two control processes, namely a *lipsd* and a *pvmd*. Conflicts may occur if the two processes are working independently. Therefore, there should exist a way to manage the activities of both processes.

One possible way is to implement a scheduler that is responsible for managing the behavior of the two processes. This approach costs too much in terms communication. There will exist a huge amount of heavy weight process-process communication within the system. Real time applications require fast response to external events. However, another solution is to deal with the two processes as threads within the same heavy weight process. This reduces the process communication cost. Moreover, it reduces the use of the UDP sockets unreliable delivery. Figure 6 shows the structure of the new heavy wait control process.

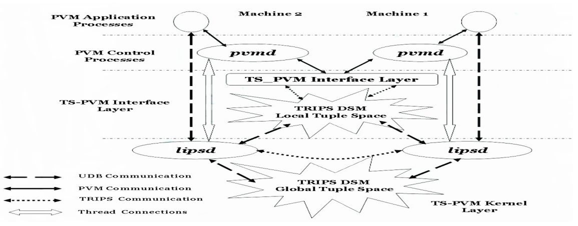


Figure 6. TS-PVM Control process structure.

The control flow in the process is organized by the scheduler. As soon as the PVM application process connects to the *pvmd* thread, it directs the request to the scheduler, which in turn forwards it to the *lipsd* thread. The *lipsd* thread may send the request to the TRIPS global DSM and/or other *lipsds*. When the required data is collected, it is returned to the *lipsd* thread that directs it to the scheduler, which in turn sends it to the *pvmd* thread. Finally, the *pvmd* thread contacts the PVM application process.

5. TS-PVM Measuring and Results

To test the performance of the proposed protocol, two types of tests have been applied. First, a fault tolerance test has been performed to test the system availability. Second, PVM send/ receive primitives performance has been measured. The measurements are performed by using four PC's. with Intel Pentium 2.4 G.H processors and 512 RAM for each. 100 Mbps Ethernet is used for processor inter-communication. The operating system in use is Windows XP professional.

5.1. Fault Tolerance Test

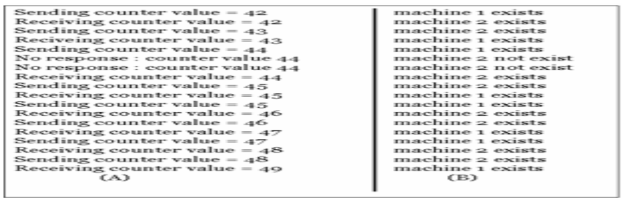
In this section, it is shown that the system tolerates with failures. This has been achieved by applying the following scenario; A client puts a counter in the system; using *pvm_send ()*. It is an entry that contains an integer. The client procedure writes the entry, another client; on another machine reads that entry; using *pvm_recieve ()*, increases the counter by 1 and then rewrites the entry with the new value. The two clients repeat these steps in a large number of iterations. While the clients are doing this process one of them is enforced to fail; by dropping his machine from the virtual machine.

```
While (true){
pvm_send(C1-id,C2_id, counter);
pvm_recive(C1_id,C2_id, counter);
counter= counter +1;
pvm_send(c2_id, c1_id, counter);
}
```

Figure 7. Availability skeleton code test.

The TS-PVM system, in this situation should wait until the failed client rejoins the virtual machine. The counter increases correctly. Figure 7 shows a skeleton code for a scheduler process that manages 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 8 shows the output of the pervious test. Part (A) shows output messages of the entry counter value while writing and reading entry. The second part of the figure (B) shows the lipsd trace output messages. In part (A), it is observed that, while the failure occurs, the counter value stops. The clients continue their operations as soon as client 2 connected again. The counter values are sequential. client 2 has been enforced to be failed and rejoined during the test.



```
Sending counter value = 42              machine 1 exists
Receiving counter value = 42            machine 2 exists
Sending counter value = 43              machine 2 exists
Reciveing counter value = 43            machine 1 exists
Sending counter value = 44              machine 1 exists
No response : counter value 44          machine 2 not exist
No response : counter value 44          machine 2 not exist
Receiving counter value = 44            machine 2 exists
Sending counter value = 45              machine 2 exists
Receiving counter value = 45            machine 1 exists
Sending counter value = 45              machine 1 exists
Receiving counter value = 46            machine 2 exists
Sending counter value = 46              machine 2 exists
Receiving counter value = 47            machine 1 exists
Sending counter value = 47              machine 1 exists
Receiving counter value = 48            machine 2 exists
Sending counter value = 48              machine 2 exists
Receiving counter value = 49            machine 1 exists
          (A)                                  (B)
```

Figure 8. Output of the availability test.

5.2. System Primitives Performance

Send only and send/receive tests for different message sizes have been performed. The results have been compared with those from the original PVM run and those from a TRIPS run. Figure 9 gives the results in case of send only. It shows the performance of the send operation in cases of PVM *(pvm-send())* and TRIPS *(out())* and the TS-PVM send. Figure 10 shows the corresponding results in case of send/receive operations.

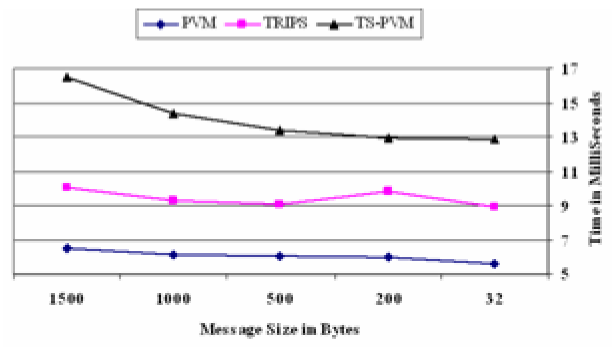


Figure 9. Send operation results.

In both cases, it is observed that the performance of the standard PVM send/receive operations is the best. Then comes the performance of the TRIPS send/receive primitives. The highest times are those of the TS-PVM protocol. These results are logic. It is a fact that the best performance among all environments is the pure message passing system PVM. Moreover, FT-PVM uses the TRIPS protocols and hence its primitives times must be larger. The TS-PVM protocol uses the primitives of both PVM and TRIPS. Therefore, it is logic to find that the time of a given operation in the proposed extension is up to the sum of times of the corresponding operations in PVM and TRIPS. The situation here is better than the worst case due to the use of threads instead of heavy wait processes for control. Otherwise, the results may be worse.

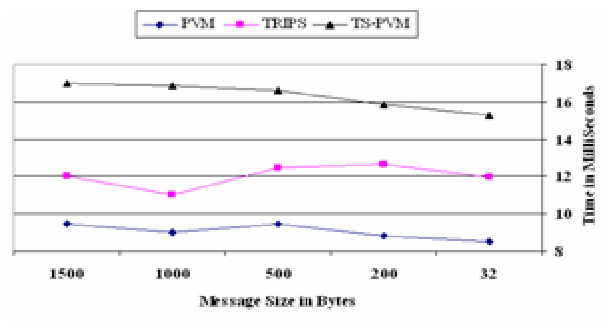


Figure 10. Send/receive operations results.

6. Conclusions

In this research work, a fault tolerant extension of PVM, that is suitable for real time applications, has been introduced. The proposed extension (TS-PVM) is based on integrating the TRIPS system fault tolerance mechanisms in PVM. Two layers have been integrated in PVM. The first layer, the interface layer, has been introduced to mask the actual primitives that are used by the protocol and to give the PVM user the ability to use the PVM original interface operations. The second layer, the kernel layer, is designed to enforce the PVM control processes, pvmds, to run under the control of the TRIPS control processes, *lipsds* that has fault tolerance capabilities. The two control processes, namely pvmd, and *lipsd,* have been integrated in one heavy weight process with a scheduler to reduce the communication costs. The proposed extension does not deal with nested configuration changes. The next step in TS-PVM, is to enable it to handle such case. The same protocol can be used for other message passing systems similar to PVM.

References

[1] Badawi A., "A Single System Image Supporting Distributed Objects," *PhD Thesis*, Cairo University, 2000.

[2] Dolev D. and Malki D., "The Transis Approach to High Availability Communication," *in Proceedings of Communication of ACM*, USA, pp. 94-102, 1996.

[3] Fatourou P. and Spirakis P., "Efficient Scheduling of Strict Multithreaded Computations," *Computer Journal of Theory of Computing Systems*, vol. 33, no. 2, pp. 173-232, 2000.

[4] Liefke T., "Extension of the Trips Prototype," *Technical Report 1.0*, University of Texas, 1998.

[5] Pruyne J. and Livny M., "Interfacing Condor and PVM to Harness The Cycles of Workstation Clusters," *Computer Journal of Future Generation Computer Systems*, vol. 12, no. 1, pp. 67-67, 1996.

[6] Rajkumar B., *High Performance Cluster Computing Programming and Applications*, Prentice Hall, 1999.

[7] Servissoglou L., Kanellopoulos E., and Kaletta D., *Fault Tolerant Message Passing under PVM,* Hermes, 1995.

[8] Setz T., Integration Von Mechanismen Zur Unterstutzung Der Fehlertoleranz in LiPS, *PhD Thesis*, University of Saarbrucken, 1996.

[9] Setz T., "Fault Tolerant Distributed Applications in Lips," *Technical Report SFB 124 09/1997*, Hamburg University*,* 1997.

[10] Stewart R. and Xie Q., *Stream Control Transmission Protocol (SCTP): A Reference Guide,* Addison Wesley, 2002.

[11] Sun Microsystems, http://java .sun.com/products /javaspaces, 2005.

[12] 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.

[13] Zarrelli R., Petrone M., and Iannaccio A., "Enabling PVM to Exploit the SCTP Protocol," *Computer Journal of Parallel Distributed Computing,* vol. 66, no. 3, pp. 1472-1479, 2008.

Usama Badawi received his Master degree in the field of object oriented databases. He has finished his PhD in distributed systems in 2001 from the technical University of Darmstadt, Germany. Currently, he is working as a lecturer in the Faculty of Science, Cairo University.