The LiPS Runtime Systems based on Fault-Tolerant Tuple Space Machines

Thomas Setz and Thomas Liefke

Technische Hochschule Darmstadt
Fachbereich Informatik
Alexanderstr. 10
D-64283 Darmstadt
Germany

{thsetz,liefke}@cdc.informatik.th-darmstadt.de

Abstract. Performing computation using networks of workstations is increasingly becoming an alternative to using a supercomputer. This approach is motivated by the vast quantities of unused idle-time available in workstation networks. Unlike computing on a tightly coupled parallel computer where a fixed number of processor nodes is used within a computation, the number of usable nodes in a workstation network is constantly changing over time. Additionally, workstations are more frequently subject to outages, e.g. due to reboots. The question arises how applications, adapting smoothly to this environment, should be realized.
LiPS[1] is a system for distributed computing using idle-cycles in networks of workstations. This system in its version 2.3 is currently used at the Universität des Saarlandes in Saarbrücken, Germany to perform computationally intensive applications in the field of cryptography and computer algebra [BMS95, LMMS94, BLZ93, WD95] on a net of approximately 250 workstations and should be enhanced to work within an environment of more than 1000 machines all over the world within the next years. The main lesson learned while using LiPS version 2.3 is that the fault as introduced by crashing machines or transient network problems should be transparent to the user. The design of version 2.4 takes this into account.
In this paper we present the runtime systems of LiPS along with performance measurements taken with the current LiPS development version 2.4.

1 Introduction

The number of machines connected to the Internet is growing by a factor greater than two every year. It is well known that the idle-time normally exceeds 70 % of the machines' uptime [LLM87]. Using the idle-time of these machines for distributed computations has many benefits which can be summarized with: "Enormous amount of additional computing power with no or only small additional investments".

Application programmers working in this environment must be provided with a programming system facilitating the development of distributed applications. This is accomplished by mechanisms shielding the programmer from the complexity of system-level programming, thus enabling him to concentrate on solving application-level problems. For example, a heterogeneous environment of different operating systems, network protocols, or processor architectures should be hidden from the programmer. Implementing a distributed application is also made more difficult by frequent changes in the availability of nodes and networks.

The LiPS system enables users to implement distributed applications in heterogeneous networks of workstations connecting machines with different processor architectures and Unix[2] operating system flavors. The system ensures that only workstations which are considered idle by their owners, are used within the distributed computations. It does not need enhanced privileges (e.g. root permissions) to perform its work. An ordinary user account is sufficient on each participating machine. The system design allows the completion of distributed computations in spite of failing nodes or network links although the performance can be compared to non-fault-tolerant systems.

[1] Library for Parallel Systems
[2] Unix is licensed exclusively through X/Open Company Limited

A programming paradigm that is suited to implement distributed computation in a heterogeneous network of workstations is the tuple space based generative communication, as originally used in the LINDA[3] programming language.

There are different approaches to integrate some level of fault-tolerance into tuple space based applications. Following [BDE94], these approaches can be divided into extensions to the tuple space runtime system [Xu 88, LX89, PTHR93, CKM92] making tuple space fault-tolerant, resilient data and processes [KS91, KS90] making tuple space and processes working on it recoverable, and transaction based or transaction style like language extensions enabling the programmer to define a sequence of tuple space operations as an atomic operation which will be evaluated completely or not at all [BDE94, BS93].

In LiPS version 2.4 we follow the approach to resilient data and processes. A more detailed description of the design and the implementation of this concept is given in [Set95].

After introducing the shared memory paradigm for parallel programming in the next section, we will explain how this is realized in the LiPS runtime systems. Finally, we will present performance measurements we have taken with the current LiPS version.

2 Generative Communication

In order to implement distributed applications, a programmer must be supplied with primitives enabling him to create additional processes or tasks, and to exchange messages among them. A conventional programming language, when augmented by interprocess communication and process manipulation primitives, is sufficient for implementing distributed algorithms. Interprocess communication (IPC) may be established accessing the network protocols, using systems like PVM [GS91], Express [Par90] or P4 [BL92]. Another approach, which is used throughout this work, is to use higher level paradigms as the tuple space based generative communication [Gel85, BCGL87, Gel88]. These approaches differ with respect to usability, efficiency, and availability on different platforms. While IPC using direct access to network protocols permits highly efficient communication, applications implemented using this approach are rather cumbersome to maintain. The generative communication approach to IPC trades efficiency against ease of use, due to the overhead introduced by tuple space management. This overhead may be kept down to a reasonable amount by analyzing communication patterns at compile-time.

This section describes the tuple space based generative communication paradigm. Using this paradigm yields elegant solutions for communication patterns typically found in distributed applications.

2.1 The Tuple Space

The tuple space is an associative, shared memory accessible to all application processes. It is called associative, as it contains data tuples, which may be retrieved addressable by their contents rather than by physical addresses, using a pattern-matching mechanism. The implementation of tuple space memory is hidden from the user and therefore may be realized on a shared-memory machine, a tightly-coupled parallel computer or on a network of workstations. Data tuples consist of a list of simple data types. We distinguish active tuples generated with the `eval()` operator from passive tuples generated with `out()`. Active tuples are used to create new threads of control within a distribute application while passive tuples are merely used to store data items. A set of operations (`in()`, `rd()`, `inp()`, `rdp()`) is used to retrieve passive tuples. Both blocking and non-blocking versions of tuple retrieval functions are available. These operations thus may be used for synchronization *and* communication tasks. The tuple extracting operations `in()` and `inp()` read a data tuple and remove it from the tuple space. If no tuple is available, the nonblocking operation `inp()` immediately returns an error as opposed to the blocking operation `in()` which suspends the calling thread until such a tuple is found. The tuple reading operations `rd()` and `rdp()` return a data tuple, again in a blocking and nonblocking manner, but do not extract the tuple from the tuple space. A more elaborate description of the tuple space can be found in [Set96].

[3] LINDA is a registered trademark of Scientific Computing Association, New Haven, Connecticut

2.2 Benefits of the Generative Communication

As the tuple space is conceptually separated from an application process, its content is not lost across thread exits. Data tuples remain available until they are consumed by some other process, which must not necessarily be around at the time the tuple is created. As a result, interprocess communication is decoupled in time. As data tuples are identified solely by their contents, and not by any other means such as senders' or recipients' process-ID, communication is made "anonymous", in that communicating processes do not need knowledge of their peer's identity. IPC using the tuple space thus decouples communicating processes logically and physically. This eases application development when compared to using a message-passing based paradigm.

As processes in a distributed application have no notion of a peer's location, migrating processes in the case that a machine becomes unavailable due to load increase or crash is made easier. A process may still retrieve messages even when it had to change to a different machine. This mechanism is transparent to the application programmer as no host addresses are involved. The tuple space communication paradigm is not tied to a particular programming language, hardware or software environment. It may thus be used for distributed applications running on a heterogeneous set of workstations. The paradigm also allows for adapting the number of usable machines, implementing what is called "adaptive parallelism" in [CFGK94, GK92]. Applications may use all available machines, shrink down to the usage of only one, and switch between these bounds of possibilities very easily. Finally, integrating the tuple space based generative communication approach into a conventional programming language requires only six additional operations.[4]

Therefore, tuple space based applications turn out to be an adequate choice for implementing distributed applications running on networks of workstations.

3 The Runtime Systems

The main problem that arises in implementing the runtime systems deals with the question of how to make a tuple space resilient to faults like machine crashes. Obviously, the solution to this problem is replication of the tuple space among different machines. This approach is implemented very efficiently in the so-called fault-tolerant tuple space machine explained later in this section.

We distinguish two fault-tolerant tuple space machines in our runtime systems. The first fault-tolerant tuple space machine implements the System Tuple Space maintaining data about the system state with processes called FixServers. The second fault-tolerant tuple space machine maintains the Application Tuple Space and is implemented with processes called MessageServers. There may exist several applications concurrently each using a private fault-tolerant tuple space machine. The System Tuple Space is shared by all applications.

A designated server process called `lipsd` resides on each machine participating in the LiPS system. The `lipsd` processes update and retrieve information from the System Tuple Space. For example, node-state information, like load of a (the) machine can be read (updated) easily through tuple space operations. `lipsd` processes update their own node-state information in the System Tuple Space in fixed intervals. A machine crash can be detected if this information is not received in time. In this case possible errors due to lost data are repaired, and watchdog mechanisms will re-integrate the crashed machine "automagically" immediately after its recovery.

Fault-tolerance on application level is implemented with a checkpointing and recovery mechanism integrated into the fault-tolerant tuple space machine. A checkpoint is correlated to the evaluation of an `eval()` operation; recovery is based on the re-execution of a failed `eval()` together with the replay of the message logging of the first execution of `eval()`. Message logging is provided via the fault-tolerant tuple space machine.

A tracing tool ala syslog(3) in co-operation with a virtual console process allows an easy level-based tracing of the LiPS-System Runtime System and Application Runtime System behavior.

The relationship between the different runtime systems described above is depicted in Figure 1.

[4] Which compares favorably to systems like Express [Par90] sporting about one thousand IPC primitives.

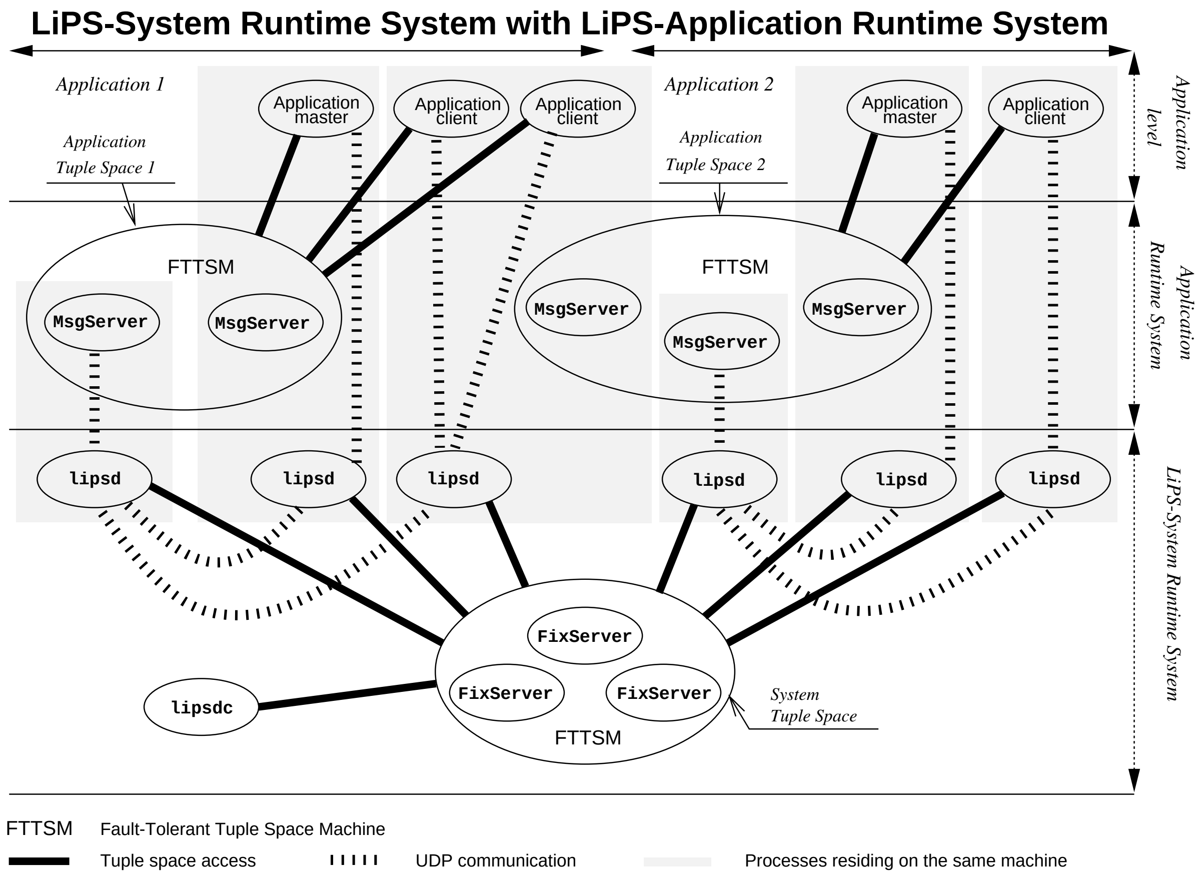

Figure 1. The different levels of the LIPS runtime systems

On each machine (gray-shaded) participating in the LIPS *system, a server process called* `lipsd` *provides access to the machine.* `Lipsd` *processes communicate through the System Tuple Space (solid line) or by exchanging UDP packets (dashed line). The Application Runtime System is based on the same communication mechanisms and uses services provided by the* LIPS*-System Runtime System.*

In this section we first introduce the design of the fault-tolerant tuple space machine being the basis for the LIPS-System Runtime System as well as for the Application Runtime System. In the following subsections we first explain the `lipsd` process in more detail and introduce a tool (`lipsdc`, the `lipsd` controller) to display the current state of the system configuration. Finally we describe the tracing facilities.

3.1 The Fault-Tolerant Tuple Space Machine

The Fault-Tolerant Tuple Space Machine replicates the content of the tuple space among several machines. If a machine that a MessageServer (FixServer) resides on crashes, the data are still available on the replicas. An additionally started server process joining the Fault-Tolerant Tuple Space Machine will be initialized with the data of an old replica. This feature makes the Fault-Tolerant Tuple Space Machine N Fault-Tolerant. In the Fault-Tolerant Tuple Space Machine every tuple is tagged with a unique ID (Sequence Number) as a result of the protocol used to replicate data across the different machines. This unique ID is used to speed up replication of events among the different servers. The protocols used in the Fault-Tolerant Tuple Space Machine are based on those given in [ADM+93, ADKM92a, ADKM92b, AAD93, Kei94]. An in-depth description of the protocol used and its implementation is given in [Set96, Set95].

As depicted in Figure 2, the tuple space is managed by several MessageServers residing on different

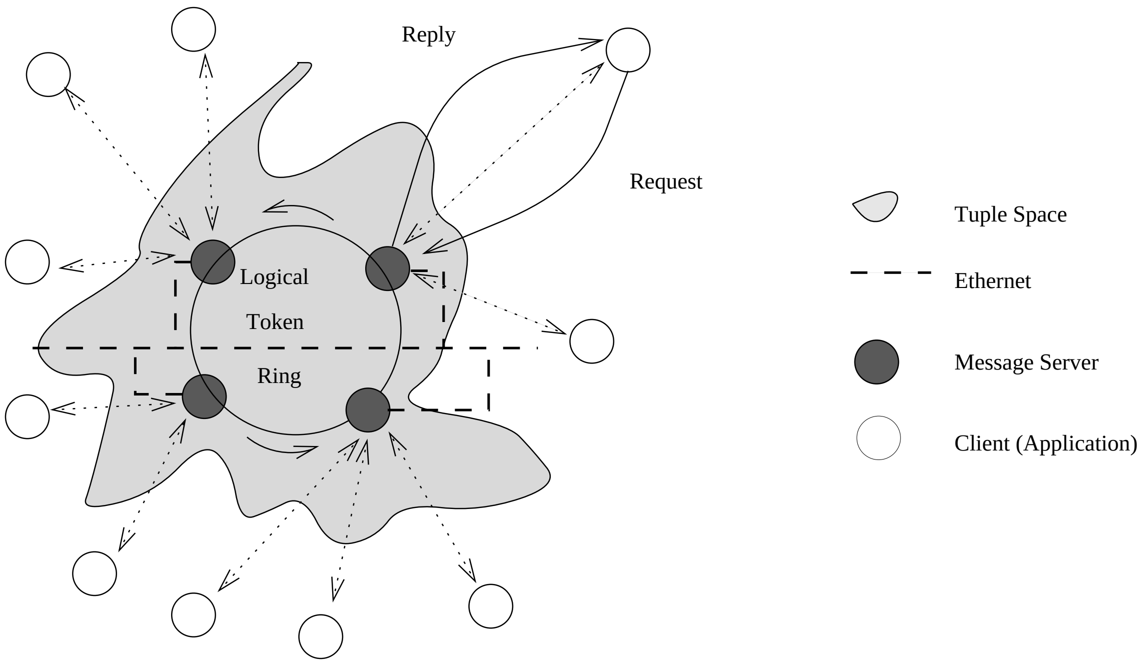

Figure 2. Processes of the Tuple Space Machine

machines. MessageServers must reside in the same broadcast domain. The broadcast facility is utilized to replicate messages very efficiently among the different servers. An additional token circulating among the servers schedules the permission to use the broadcast facility - avoiding Ethernet saturation due to collisions. The circulating token ships additional data enabling among other things flow control between the replicas. Additionally, each broadcast message (tuple) is tagged in sequence with a unique ID. This procedure establishes a linear order among the tuples of the Fault-Tolerant Tuple Space Machine and speeds up replication.[5]

The crash of a MessageServer is detected if the circulating token is not received in time. In this case several protocol layers are automatically triggered which brings back the Fault-Tolerant Tuple Space Machine into an operational state.

As shown in Figure 2, an application process sends requests to the MessageServer which is assigned to it. A request can either contain a tuple or a template. In the following, we first explain how MessageServers process a tuple, and second how templates are processed.

As the MessageServers share the same broadcast domain, a MessageServer is able to broadcast the tuple and hence replicate it on multiple MessageServers with only one physical operation. At any time only one MessageServer may broadcast a tuple, namely the MessageServer holding the token message. After a MessageServer has finished broadcasting messages (tuples), it sends the token to the next MessageServer. With respect to this token transfer, the MessageServers form a logical ring. Messages being broadcast are tagged with a unique sequence number. The sequence number of the last broadcast message of a MessageServer is sent within the token. The next MessageServer intending to broadcast knows the sequence number of the last broadcast message and continues the sequence, thereby establishing a total order on the messages broadcast. Within one token rotation, several tuples may be broadcast by each MessageServer.

If a MessageServer receives a template, it first tries a match on its local tuple space. If no tuple matching the template is found, the MessageServer notifies the requesting application process (NACK). Otherwise, if the MessageServer finds a match, it must first synchronize with the other MessageServers. In order

[5] If a broadcast message was not received on a replica this circumstance is easily obtained, as there is a gap in the sequence of received messages. In this case a retransmission could be requested immediately.

to notify the other MessageServers of the tuple access, it is sufficient to send the sequence number (4 bytes), the application process accessing the tuple and the event number in its message logging (4 bytes) as well as the type of access (1 byte) to identify the operation to the replicas. These items of access information now are added to the circulating token. The size of the token then determines the number of reading and extracting tuple space operations which may be replicated within one token rotation.

3.2 The LiPS Daemon `lipsd`

A system server process called `lipsd` resides on each machine in a LiPS installation. This process gives the LiPS system access to the machine and is responsible to obey the idle-time restrictions for the machine this particular `lipsd` is residing on. Furthermore, the `lipsd` processes provide services such as starting and controlling application processes. Each `lipsd` updates the node state information of its machine held in the System Tuple Space within a given time. If a `lipsd` fails to update this information within the given period of time, the machine it resides on is assumed to have crashed. In this case, all other `lipsd` processes are informed of the crash of the machine, and possibly corrupted data in the System Tuple Space are repaired. Additionally, all application processes residing on that machine are scheduled to be restarted on another machine. Informing all `lipsd` processes of this event is realized with a signal indicator shipped with every tuple space operation. As every `lipsd` updates its node-state information periodically, it will receive the signal indicator soon after the crash of a machine. On receipt of the signal indicator, the `lipsd` process triggers appropriate signal handler functions. The 'first' `lipsd`, finding himself in the signal handler, will repair possibly corrupted data structures (lost tuple) and if necessary will schedule lost application processes to be restarted. All other `lipsd` processes only update their locally cached data. The 'first' `lipsd` mentioned above is identified with the help of an automatically (Fault-Tolerant Tuple Space Machine) generated tuple which will be destructively read (`inp()`) by the 'first' `lipsd` and is not available to the other `lipsd` processes (`inp()` returns an error). Within the `lipsd` process all destructive read operations (`in(), inp()`) are immediately followed by a tuple generating operation (`out()`). This eases the restoring of corrupted data after a crash as the Fault-Tolerant Tuple Space Machine simply re-injects the tuple into the System Tuple Space if the last tuple space operation of a crashed `lipsd` has been a destructive read. These mechanisms enables us to handle recursive crashes of `lipsd` processes.[6]

As already mentioned, each `lipsd` process maintains a tuple for its machine in the System Tuple Space which contains the node state information. This information is based on the node's status (running, idle etc.), number of users, load average, number of application processes and the time of the last update of this tuple. This information is considered when starting new application processes to achieve a well-balanced process and load distribution. In order to detect failing application processes, the System Tuple Space holds a table for each running application process. When this information gets updated by a `lipsd`, this old table is compared with the new information. The processes which are marked as running within the old table but marked as non-running in the new one, need to be restarted. These data are periodically read by the applications' Fault-Tolerant Tuple Space Machine which then in turn is able to request a restart of crashed application processes.

As the `lipsd` processes are permanently running on all machines in a LiPS installation, in particular even if a machine is unavailable to run a LiPS application process due to idle-time restrictions, they are implemented such that they cause minimal overhead for the machine. In the first place, the runtime and memory requirements are minimized e. g. in the case that a machine is idle, the load is updated every 15 seconds, otherwise every 5 minutes. As the LiPS system was used in relatively fast networks so far, the network overhead was not optimized yet. When using LiPS in a huge installation or in wide-area networks, more attention must be paid to reducing the communication traffic.

3.3 The LiPS Daemon Controller `lipsdc`

The `lipsdc` tool can be used for two purposes: on one hand it is the means to initialize the global data structures of the LiPS-System Runtime System in the System Tuple Space during the start-up of a LiPS configuration. On the other hand, it enables the administrator of a LiPS installation to

[6] i.e., if a `lipsd` crashes while it was recovering the data for a formerly crashed `lipsd`.

interactively change the configuration during runtime and provides users with the information about their application processes in the LiPS system.

If used during the start-up of a LiPS configuration, a number of global tables are created in the form of tuples in the System Tuple Space. Moreover, a configuration file is read containing entries for each machine becoming a member of the LiPS installation such as the machine architecture or a specification of what it means for that machine to be idle. Finally, several configuration files are created and updated resp. which have to be copied together with the architecture-dependent `lipsd` binary to each machine taking part in the newly set-up LiPS installation. These files hold data about the FixServers' ports, the process IDs for the different `lipsd` processes and trace level data.

In the interactive mode, the `lipsdc` allows to display and change the current LiPS configuration such as request the load state of each machine, add or remove machines in the installation, change the idle-time specifications, or display all application processes including information about the user to which each process belongs or which machine each process is distributed on within the network.

3.4 Tracing LiPS

It is very difficult to follow the execution of a program in a distributed environment due to the fact that there are actions taking place concurrently in different processes possibly residing on different processors. The LiPS runtime systems provide the user with a facility to trace the output of the applications running within it in a terminal and/or a file. The same mechanism is applicable to the LiPS system itself.

Tracing is based on a syslog(3)-like macro call in the source code. Depending on the actual tracing level of the module, the macro output is redirected to a virtual console, identified by an Internet address and a port on which the console process is listening, or discarded. The tracing levels of the different modules are defined in a file. The content of this file is reread on receipt of a signal (SIGTRAP) and the behavior of the tracing tool changes according to the newly read configuration. In other words, the tracing behavior of LiPS applications as well as the LiPS system itself is (at runtime) adjustable on a per-module basis. Additionally, it is possible to disable tracing with the same mechanisms to gain full performance, and to switch between these bounds of possibilities very easily.

Besides simple displaying of this tracing information in lines, it can also be used for profiling the distributed programs. As the output can optionally be generated in the PICL format (Portable Instrumented Communication Library) [GHPW90], a profiler compatible with this format can be easily integrated with LiPS.

4 Performance

In this section we present some runtime measurements for tuple space operations with a number of application processes concurrently accessing the tuple space. The test is based on the benchmark given in [Mat95]. In this report the tests were performed using PVM Version 3.2.6, p4, TCGMSG and SCA Linda on a network of IBM RS/6000 model 560 workstations under the operating system AIX 3.2. In those measurements, the best benchmark timing of different tests suites was used to compare the different systems because the network on which the tests were made was concurrently used by other users and therefore, the test conditions were not equal. Our tests were made within another environment and for this reason, the timings presented here are not really comparable to the timings taken from [Mat95]. Anyway, the timings presented here show that they are in the same order of magnitude. Keeping in mind that we are comparing a 'fault-tolerant design' with non fault-tolerant ones, this is an impressive result.

In this section we first describe the test environment used and give the skeleton code for the ping/pong test performed. Then we show the timing for one ping/pong operation using 1 to 3 MessageServer processes and up to 24 clients.

4.1 Our Test Environment

In our tests, the MessageServers resided on Sun SPARCstation 20's with 128 MB main memory running the operating system SunOS 4.1.4. The replication of messages was done in a 10 Mb/s Ethernet. The application processes resided on Sun SPARCstations SLC and ELC with a main memory of 16–64 MB running the operating system SunOS 4.1.3. The connection to those was made via an ATM-LAN concentrator with an ATM switch having a theoretical bandwidth of 155 Mb/s.

4.2 Ping/Pong Test

In the ping/pong test, the time needed by an application process to write a tuple into the tuple space and read the same tuple afterwards is measured. The loop in which those operations are executed is the following:[7]

```
for (iters = iterations; iters-- ; ) {
    t0 = wtime();

    out("ping", "al", buffer, buffer_size);
    in("ping", "AL", buffer, &buffer_size);
    *tp = wtime();
    *tp++ -= (t0 + twtime);
}
```

The function `wtime()` calculates the time before and after an `in()` – `out()` sequence. The average time for one iteration is calculated and the best results are taken. The tests have been executed using tuples each with a size of 800 bytes. The size of a tuple is determined by the array `buffer`, the size of which is specified in `buffer_size`.

4.3 Results

The runtime for a different number of application processes and a different number of MessageServers is given in Figure 3 whereas Table 1 shows the runtime of one application process and one MessageServer. In the last case, both the MessageServer and the application process resided on Sun SPARCstation 20's.

As mentioned earlier, no absolute results can be obtained from the different test environments. However, the tests show that the approach presented in this paper achieves comparable results of the same order. Thus, a similarly efficient communication is realized for fault-tolerant applications.

System Name	Runtime	fault-tolerant
LiPS	5.8 ms	yes
TCGMSG	2.3 ms	no
P4	2.3 ms	no
PVM	2.8 ms	no
SCALinda	3.4 ms	no

Table 1. Runtime for one application process and one MessageServer

[7] The second parameter of the `out()` and `in()` call define the data type of the used variables — array of long in this case. Upper-case letters identify formal parameters; lower-case letters identify actual parameters. `twtime` stands for the correction of the duration of a `wtime()` call.

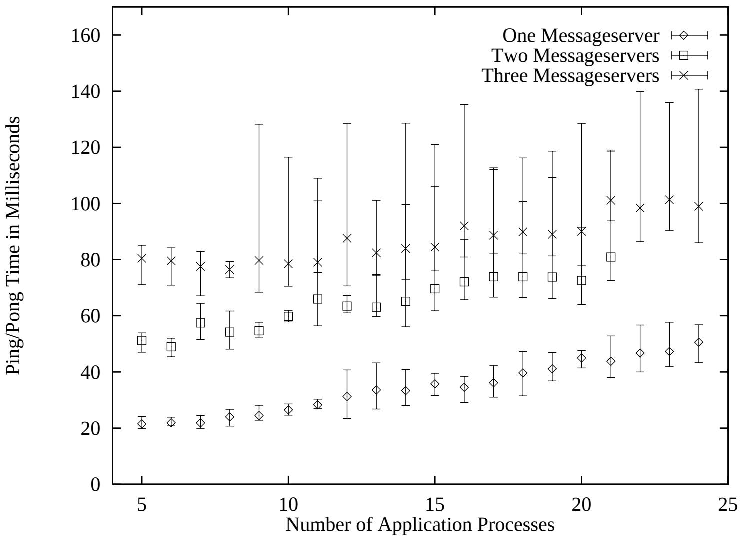

Figure 3. Runtime of the Ping/Pong test for tuple space operations with 5 – 24 application processes

Displayed are best (bottom line), worse (top line) and average values (diamond/box/cross).

5 Conclusion and Summary

LiPS is a system using the idle-time in a heterogeneous network of workstations for distributed applications. It provides its users with the tuple space based generative communication paradigm of distributed computing. The former version of LiPS has proven to work within an environment of about 250 machines. The main problem arising while designing a runtime system within this environment is dealing with faults caused by crashing machines or transient network errors. The design for Version 2.4 takes this into account and provides mechanisms to tolerate such faults.

The design is based on a Fault-Tolerant Tuple Space Machine being able to maintain the shared memory in spite of crashing machines. A dedicated process resides on each node participating in the LiPS system, the so called `lipsd`, building the interface from the LiPS system to the machine. `lipsd` processes communicate through a private tuple space called System Tuple Space which is realized with a Fault-Tolerant Tuple Space Machine. The server processes running the System Tuple Space are called FixServer. FixServer and `lipsd` processes constitute the LiPS-System Runtime System providing services to the Application Runtime System. The Application Runtime System is also based on a Fault-Tolerant Tuple Space Machine. To differentiate the servers for the Application Tuple Space Machines from the system processes, they are called MessageServers.

The recognition of machine crashes (or network failures) is based on a timeout for periodic updates of a node's state (`lipsd`) in the System Tuple Space. In case of a crashed machine, application processes which resided on it are easily detected and their restart is scheduled. Propagation of these data is triggered through a signal mechanism added to the tuple space operations. Fault-tolerance on application level is based on checkpoints and message logging. Both mechanisms are integrated into the Fault-Tolerant Tuple Space Machine.

The performance measurements show that the LiPS runtime systems are favorably comparable with similar systems, and additionally provides fault-tolerance.

Bibliography

[AAD93] Amir Y., Amir O., and Dolev D. A Highly Available Application in the Transis Environment. In *Proceedings of the Hardware and Software Architectures for Fault Tolerance, LNCS 774*, 6 1993.

[ADKM92a] Amir Y., Dolev D., Kramer S., and Malki D. Membership algorithms for multicast communication groups. In *Intl. Workshop on Distributed Algorithms proceedings (WDAG-6)*, 11 1992.

[ADKM92b] Amir Y., Dolev D., Kramer S., and Malki D. Transis: A communication sub-system for high-availability. In *Annual International Symposium on Principles of Distributed Computing*, 7 1992.

[ADM+93] Amir Y., Dolev P., Melliar-Smith P., Agarwal D., and Ciarfella P. Fast Message Ordering and Membership using a Logical Token-Passing Ring. In *13th International Conference on Distributed Computing Systems (ICDCS)*, number 13 in IEEE, pages 551–560, Pittsburgh, 5 1993.

[BCGL87] Bjornson R., Carriero N., Gelernter D., and Leichter J. Linda the Portable Parallel. Technical Report YALEU/DCS/TR-520, Yale University, Department of Computer Science, New Haven, 2 1987.

[BDE94] Bakken D. E. *Supporting Fault-Tolerant Parallel Programming in Linda.* PhD thesis, The University of Arizona, 6 1994. Department of Computer Science.

[BL92] Butler R. and Lusk E. Users Guide to the p4 Parallel Programming System. Technical Report ANL-92/17, Argonne National Laboratory, 5 1992.

[BLZ93] Buchmann J., Loho J., and Zayer J. An Implementation of the General Number Field Sieve. In *Proceedings of Crypto'93*, Heidelberg, August 1993. Springer Verlag.

[BMS95] Buchmann J., Müller V., and Shoup V. Distributed Computation of the Number of Points on an Elliptic Curve over a Finite Prime Field. Technical report, Universität des Saarlandes, SFB 124 TP D5, 03/95, 1995.

[BS93] Bakken D. E. and Schlichting R.D. Supporting Fault-Tolerant Parallel Programming in Linda. Technical Report 93.18, Department of Computer Science, The University of Arizona, 6 1993.

[CFGK94] Carriero N., Freeman E., Gelernter D., and Kaminsky D. Adaptive Parallelism and Piranha. Technical Report YALEAU/DCS, Yale University Department of Computer Science, 2 1994.

[CKM92] Chiba S., Kato K., and Masuda T. Exploiting a weak concistency to implement distributed tuple space. In *Proceedings of the 12th International Conference on Distributed Computing Systems*, 6 1992.

[Gel85] Gelernter D. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[Gel88] Gelernter D. Getting the Job Done. *Byte*, 11 1988.

[GHPW90] Geist G.A., Heath B.W., and Peyton W. and Worley P.H. *PICL a Portable Instrumented Communication Library.* ORNL/TM-11130, C-Reference Manual edition, 7 1990.

[GK92] Gelernter D. and Kaminsky D. Supercomputing out of Recycled Garbage: Preliminary Experience with Piranha. *Sixth ACM International Conference on Supercomputing*, July 1992.

[GS91] Geist G. A. and Sunderam V. S. The PVM System: Supercomputer level concurrent computation on a heterogenous network of workstations. In *Proceedings of the Sixth IEEE Distributed Memory Computing Conference*, 3 1991.

[Kei94] Idi Keidar. *A Highly Available Paradigm for Consistent Object Replication.* Master thesis, Hebrew University of Jerusalem, Institute of Computer Science, 4 1994.

[KS90] Kambhatla S. Recovery with limited replay: Fault-tolerant processes in Linda. Technical Report CS/E 90-019, Department of Computer Science, The Oregon Graduate Institute, 9 1990.

[KS91] Kambhatla S. *Replication issues for a distributed and highly available Linda tuple-space.* Master thesis, Oregon Graduate Institute, Department of Computer Science, Beaverton, Oregon, 9 1991.

[LLM87] Litzkow M.J., Livny M., and Mutka M.W. CONDOR - A Hunter of idle Workstations. Technical Report 730, University of Wisconsin - Dept. of Computer Science, Madison, December 1987.

[LMMS94] Lehmann F., Maurer M., Müller V., and Shoup V. Counting the Number of Points on Elliptic Curves over Finite Fields of Characteristic greater than three. In *Proceedings of ANTS I*, 1994.

[LX89] Liskov B. and Xu A. A design for a fault-tolerant, distributed implementation of Linda. In *Proceedings of the Nineteenth International Symposium on Fault-Tolerant Computing*, 6 1989.

[Mat95] 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.

[Par90] ParaSoft Corporation, CA. *Express C Reference Guide Version 3.0*, 1990.

[PTHR93] Patterson L. I., Turner R. S., Hyatt R.M., and Reilly K. D. Construction of a fault-tolerant distributed tuple-space. In *Proceedings of the 1993 Symposium on Applied Computing.* ACM/SIGAPP, 2 1993.

[Set95] Setz T. LiPS*; Manual Version 2.4*, 10 1995. Universität des Saarlandes, Fachbereich Informatik, Lehrstuhl Prof. Buchmann.

[Set96] Setz T. *Integration von Mechanismen zur Unterstützung der Fehlertoleranz in* LiPS. Phd thesis, Universität des Saarlandes, 2 1996. Fachbereich Informatik, Lehrstuhl Prof. Buchmann.

[WD95] Weber D. An Implementation of the Number Field Sieve to Compute Discrete Logarithms mod p. In *Advances in Cryptology Eurocrypt 95.* pp. 95–105, 9 1995.

[Xu 88] Xu A. *A Fault Tolerant Network Kernel for Linda.* Master thesis, MIT, Laboratory for Computer Science, Cambridge, 8 1988.

This article was processed using the LaTeX macro package with LLNCS style