View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Acronym

Software Fault-Tolerant Distributed Applications in LiPS

Thomas Setz

Keywords: hypercomputing, software fault-tolerance, Linda, idle-time, recovery line

Abstract This paper illustrates how software fault-tolerant distributed applications are implemented within LiPS version 2.4, a system for distributed computing using idle-cycles in networks of workstation.
The LiPS system [SR92,SR93,STea94,Set95,SF96,ST96,SL97,ST97] employs the tuple space programming paradigm, as originally used in the Linda[1] programming language. Applications implemented using this paradigm easily adapt to changes in availability as they occur in workstation networks. In LiPS, applications are enabled to terminate successfully in spite of failing nodes by periodically writing checkpoints, freezing the state of a computational process, and keeping track of messages exchanged between checkpoints in a message log. The message log is kept within the tuple space machine implementing the tuple space and replayed if an application process recovers. This assumes deterministic behavior of the application process but allows independent checkpoint generation and alleviates the need for application-wide synchronization in order to generate sets of consistent checkpoints.

1 Overview

Workstation computers are becoming increasingly popular due to their high performance/cost ratio. With increasing numbers of workstations and the advent of high-speed networks, supercomputer-like aggregate computational power is available and, as shown in [BLZ93,LMMS94,Web95,BMS95], can be used to perform useful computations.

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

[1] Linda is a trademark of Scientific Computing Association., New Haven, Connecticut

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.

Transparency, meaning hiding the physical implementation of a distributed application, is the utmost goal of every distributed programming system.

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 users are used within the distributed computations. The system also guarantees successful completion of distributed computations in spite of failing machines or network links. Within the last years, LiPS has been used to distribute computations on about 250 workstations connected to the campus network at the University of Saarbrücken (Germany) and will be enhanced to distribute applications on more than 1000 machines within the next years.

This paper presents some basic decisions taken when designing LiPS version 2.4. This version supports a software-fault-tolerant generative communication paradigm based on the tuple space, as introduced by the coordination language Linda [GCCC85].

The next chapter contains an introduction to generative communication, a programming paradigm suited to implement distributed computations in networks of workstations. Then, an introduction to the terminology used for coping with fault-tolerance is given along with the model of software failure patterns used throughout this paper. Next, the concept of software fault-tolerance is introduced. It permits grouping software components and strategies into layers thus supplying general distributed applications with a variety of software fault-tolerance mechanisms. This section also illustrates different methods to restart single crashed processes within the application framework and discusses the benefits of our approach. The general

[2] Unix is licensed exclusively through X/Open Company Limited

concept of software fault-tolerance is then applied to distributed applications implemented along the generative communication paradigm. Using layer-3 software fault-tolerance enables the user to implement fault-tolerant applications. By relaying all communication activities via the tuple space, a complete log of all messages exchanged between application processes is available in the tuple space machine, even while individual application processes are prone to failure. Having access to a complete history of messages exchanged per process permits recovering application processes in a highly efficient manner, but this requires a well-suited system design. The last section presents the design of our Fault-Tolerant Tuple Space Machine along with its integration into the LiPS system.

2 Related work

There are different approaches to integrate different levels 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,CKM92,PTHR93] making tuple space fault-tolerant, resilient data and processes [KS90,KS91] 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].

3 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 inter-process 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.

3.1 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 distributed 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. Hence, these operations 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 non-blocking 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 non-blocking manner, but do not extract the tuple from the tuple space. A more elaborate description of the tuple space can be found in [Set96].

3.2 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, inter-process 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 peers' 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 has 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.[3]

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

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

4 Failure Models

Workstation computers are prone to failures. As a consequence, this may lead to failures in applications implemented using the LiPS system. Several failure patterns can be distinguished:

- Crash-failures or as called in [SS83] "fail-stop-processors", are observed when a machine halts on an error condition, forcibly terminating all application processes local to the processor affected.
- Soft-fail-stop-failures are observed when a machine stops on an error, terminating all local application processes. But there exists storage, possibly residing on another unaffected machine which remains intact and is accessible.
- Omission-failures are observed when machines sometimes fail to send or receive messages.
- Byzantine failures, where machine start sending wrong and even contradictory information as a result of an error.

The LiPS system is able to cope with soft-fail-stop failures. Data that should remain accessible in spite of machine failures is kept in a storage called repository. The system further deals with omission-failures as messages are exchanged using the UDP protocol of the TCP/IP protocol suite[4]. Handling Byzantine failure is rather expensive, and these failures are rarely observed in practice. Therefore, we will not consider Byzantine failures in this work.

5 Software Fault-Tolerance

Distributed applications are usually based on fault-tolerance mechanisms provided by a node operating system. The term "software fault-tolerance", as introduced

[4] This protocol implements a "best-effort" delivery. Datagram messages may be lost or duplicated by the underlying network layers.

in [YC83], is used to subsume methods and software components responsible for detecting and correcting errors causing a distribute application to crash or hang, that are not already handled in the underlying operating system. Software fault-tolerance may be organized in layers – Figure 1 gives an overview. Layers are discriminated along the levels of availability and data consistency.

Normally, distributed applications are based on the services delivered by the node operating system, the so-called level 0 of software fault-tolerance. If a node crashes, manual intervention is required to restart the processes which were residing on that node. Shared data may be lost or left in an inconsistent state.

Layer-1 software fault-tolerance is reached by providing for automatic restart of application processes in the event of a crash. This layer provides for enhanced application availability, as no manual intervention is required for the entire application to complete. Restarted processes still need to re-do their entire computation, resulting in a complete loss of effort spent on the previous run. Abort of a single process may force the entire application to halt if shared global data is left in an inconsistent state, thus wasting the entire time spent computing so far.

Layer-2 software fault-tolerance requires application processes to create checkpoints capturing a process's state. If an application process crashes, it can be restarted from its latest checkpoint, thereby reducing run time spent as effort to reach the state at crash time. Furthermore, messages sent and received in the interval between checkpoint generation are kept in a message log. If an application process restarts from a checkpoint, it will receive the same set of messages it got on its initial run and therefore will compute the same results again. This requires computations to be fully determined by received input messages. Restarting processes from an earlier checkpoint constitutes a backward error recovery strategy. An application process is said to be in "recovery" state if it has not yet reached the state at crash time. It is said to be "active" resp. "operational" if its computation proceeds beyond the crash state. Layer-2 software fault-tolerance strategies lead to increased application process availability, as well as increased message-space consistency.

A distributed application is said to be layer-3 software fault-tolerant if data kept in a file system are recoverable after a failure. If a process is restored from a checkpoint

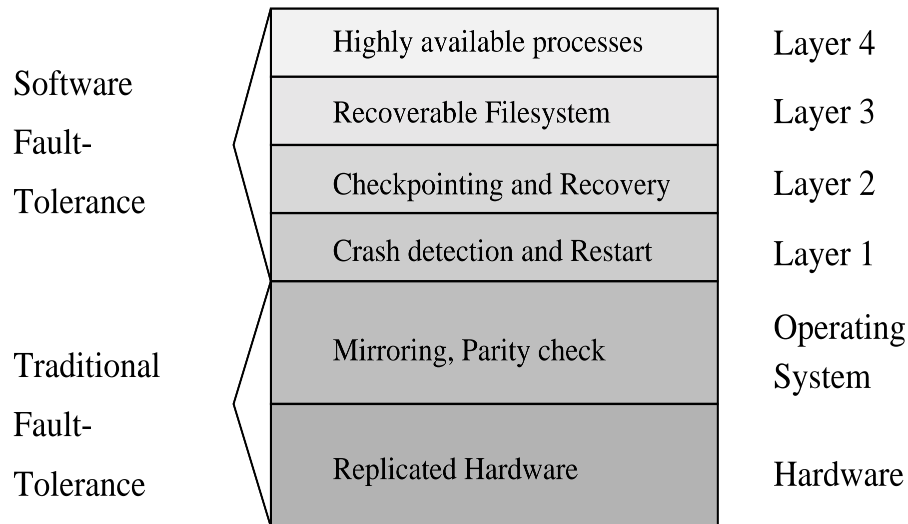

Figure1. Layering of Software Fault-Tolerance Strategies

image, all files that were open at crash time should be accessible even if the process was restarted on another machine. Changes made to the files must be un-done prior to restarting from a checkpoint. Level-3 software fault-tolerance increases data consistency of applications and increases process availability as processes are able to migrate to another machine.

Layer-4 software fault-tolerance mechanisms are used if an application needs a very high availability. This is accomplished by replicating several copies of each application process on different nodes. When a process instance fails, identical output is available from another instance of this application process. Thus, the application continues to perform its computation apart from the time it takes to notice node failure, and to use results produced by another process instance. All it takes to implement layer-4 fault-tolerance is to synchronize replica behavior. Layer-4 software fault-tolerance increases process availability.

6 Recovery Design Alternatives on the Application Level

If a process of a distributed application has to be started from its last checkpoint, the question arises how to treat messages sent or received by the process since its last checkpoint. If e.g. process X in Figure 2 crashes, it may be restarted from

checkpoint x_3 without affecting other processes belonging to the application. However, if process Y crashes at time $t = 14$ after sending message m, it will generate and re-send message m to process X when restarted from checkpoint y_2.

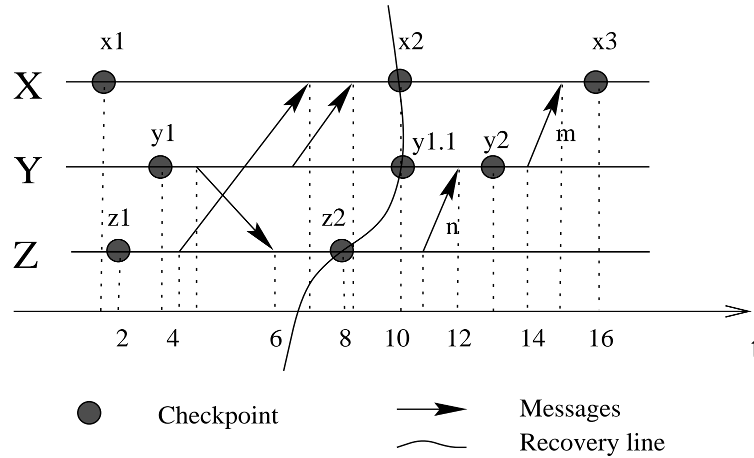

Figure2. Recovery

There are two basic alternatives for dealing with this problem. The first one, later referenced as Backward Backward Error Recovery (BBER), involves undoing all effects caused by a process in the time interval between its last checkpoint and the time of the crash. To undo the effects caused by a failed process on another active process, the failed process will be rolled back into an earlier state, and the other process will be restarted from a checkpoint too. Consider process Y failed after sending message m in the example. The BBER strategy would then require restarting process X from checkpoint x_2 to undo the effects of re-sending message m after Y is restarted from checkpoint y_2. The second alternative, later referenced as Backward Forward Error Recovery (BFER), ensures that a process restarted from a checkpoint executes the same instructions as on its initial run. However, effects affecting other processes are suppressed. Applied to the example, process Y would be restarted from checkpoint y_2. When restarted, Y will again generate and send m. Duplicate reception of m by X must then be suppressed by some external means.

The BBER may lead to a "domino effect", requiring the restart of other processes indirectly affected by a process abort. If process Z crashes after sending n, X, Y, and Z would need to be restarted from their respective checkpoints x_1, y_1, and z_1, as they received some messages sent by Z after writing its latest checkpoint image. Within this context, messages n and m are called "orphan messages". Orphan messages may lead to a domino effect which possibly affects all application processes[5]. Applying the BBER strategy requires careful scheduling of checkpoints in order to avoid orphaned messages. In the best case there is no information flow at the time all application processes create a checkpoint image. This could be accomplished by scheduling process Y to write its checkpoint image $y_{1.1}$ at time $t = 10$. At this point in time, no unreceived messages are present in the system. If process Z would crash after sending n, restarting processes X and Y from checkpoints x_2 and $y_{1.1}$ would be sufficient to undo all changes made by process Z, which would then be restarted from checkpoint z_2. Checkpoints x_2, $y_{1.1}$ and z_2 are said to constitute a "recovery line", or "strongly consistent checkpoint". The drawback is that all processes must be considered when writing checkpoint images for every single application process. This requires synchronization among all processes in order to determine whether it is safe to write a checkpoint image.

Applying a BFER strategy alleviates the need for synchronization prior to taking checkpoints. Individual processes may write checkpoint images at any time. This requires keeping the message log in some entity surviving the process crash which is responsible for supressing orphaned messages and replay of already received messages.

7 Combining Generative Communications and Software Fault-Tolerance

This section shows how software fault-tolerance mechanisms are added to programs based on the generative communication paradigm. Applying layer-3 methods yields an acceptable level of fault-tolerant execution for such applications. Application of

[5] There are more problems with BBER. An in-depth treatment is given in [MN94]

layer-3 strategies to these distributed programs is then examined in greater detail. As all inter-process communication is done via the tuple space, there is already a system entity in place to keep the message log for each application process, unaffected by application process crashes. This lends itself to using the BFER strategy for process recovery.

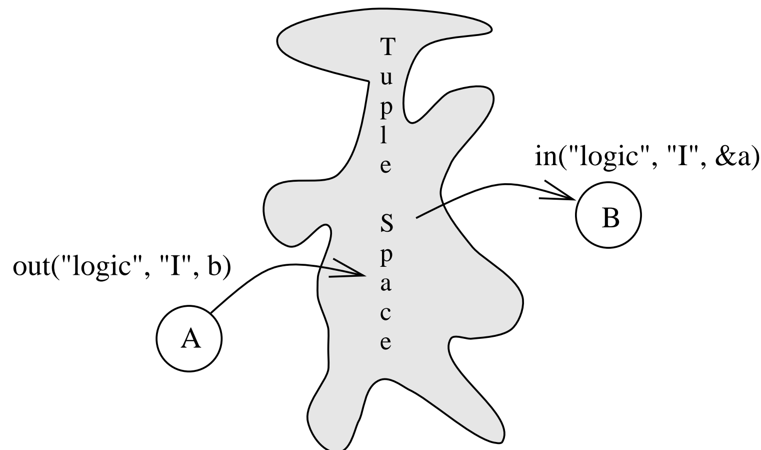

Figure3. IPC using the Tuple Space

On each machine where application processes are to be executed, a system service program is installed. Its task is to control and to restart application processes in the event of a machine crash. Thus layer-1 software fault-tolerance is reached. The implementation of these system service processes is described in greater detail in [Fis96]. Layer-1 software fault-tolerance by itself is not sufficient for fault-tolerant execution of applications using the generative communication approach for inter-process communication as shared data may be left in an inconsistent state.

Layer-2 software fault-tolerance adds checkpoints and message logging to the layer-1 software fault-tolerance mechanisms. Applications implemented using the generative communication approach for IPC are already exchanging messages by adding and removing data tuples from a global tuple space, as depicted in Figure 3. Tuple space operations are kept in a per-process log. For each process, its checkpoint image freezes the state of the particular computation performed by the process.

Messages sent or received as the checkpoint generation are commonly referred to as "events" and are also kept in the message log. Figure 4 shows the message log after exchanging messages in Figure 3. The call to `out()` in Process A is uniquely identified with event number 2. We use the BFER in order to re-integrate a crashed process into an application. It is sufficient to restart an application process from its latest checkpoint image and to supply messages from its message log. In particular, duplicate output messages may now be identified and are suppressed. When a process succeeded in taking a checkpoint image, events prior to the checkpoint event may be discarded from the message log. In Figure4, event 4 for process A would no longer be present in process B's message log as it is already incorporated into its checkpoint taken at event 7; process B would not receive this message again when restarted from this checkpoint. However, when process A is restarted from scratch after failing after event 5, the output message generated at event 2 must be prevented from reaching the tuple space, as this would create a duplicate tuple. Processes are said to be in recovery state when their communication is screened by a message log.

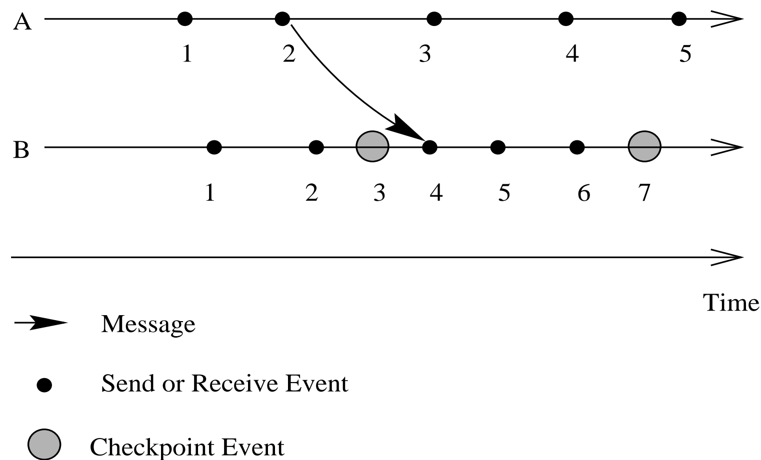

Figure4. Message Logging

Distributed applications may need to access large amounts of data kept in files. If a machine fails and becomes unavailable, data kept on this machine is lost and

may cause the entire application to fail. Layer-3 software fault-tolerance addresses this problem. It ensures that the file system environment[6] may be restored when encountering an error. Layer-3 software fault-tolerance may be implemented by replicating files accessed by application processes. If a process is to be restarted, all files required by the process have to be copied to its working directory prior to the process restart.

Normally, there is no need for application processes to be highly available. Applying layer-4 software fault-tolerance strategies is not necessary as applications are able to run to completion if layer-3 software fault-tolerance mechanisms are applied. Replicating individual application processes in order to gain increased availability would consume additional computing power which could be used by other application processes too.

8 The LIPS System

The main problem that arises in implementing a software fault-tolerant system for applications based on the generative communication paradigm deals with the question of how to make the 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 the LIPS system. The first Fault-Tolerant Tuple Space Machine implements the System Tuple Space maintaining data about the system state e.g. which machine is idle. The second Fault-Tolerant Tuple Space Machine maintains the Application Tuple Space. Figure 5 on page 15 gives an overview.

There may exist several applications concurrently each using a private Fault-Tolerant Tuple Space Machine. The System Tuple Space is shared by all applications.

[6] The file system environment of an application consists of all files being accessed by an application process. Processes are expected to access files present in their working directories; in particular, no file may be open concurrently by several processes

In this section, we first introduce the different components of the runtime systems of LiPS and their cooperation. A more detailed explanation is given in [SL97]. The design and implementation of the Fault-Tolerant Tuple Space Machine is explained next. A detailed description is given in [Set96].

8.1 The LiPS Runtime Systems

We distinguish two different runtime systems within LiPS. The system runtime system and the application runtime system. Both are based on a Fault-Tolerant Tuple Space Machine. The server processes of the Fault-Tolerant Tuple Space Machine for the system runtime system are called FixServer; those of the application runtime system's Fault-Tolerant Tuple Space Machine MessageServer. The relationship between the different runtime systems described above is depicted in Figure 5.

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. A more detailed description of these mechanisms is given in [SF96].

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.

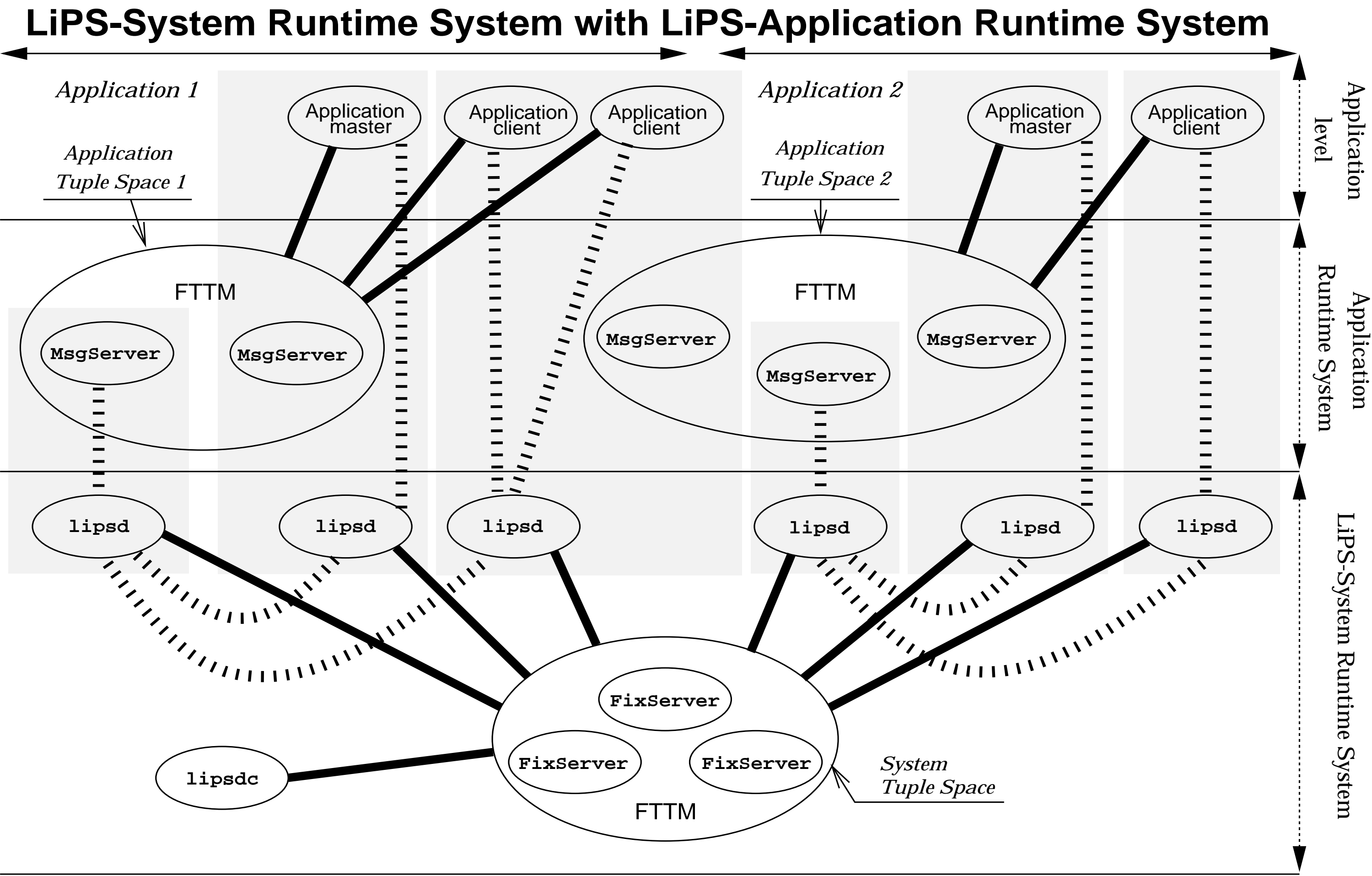

Figure5. The different levels of the LiPS runtime systems

8.2 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]. An in-depth description of the protocols used and their implementation is given in [Set96,Set95].
As depicted in Figure 6, the tuple space is managed by several MessageServers

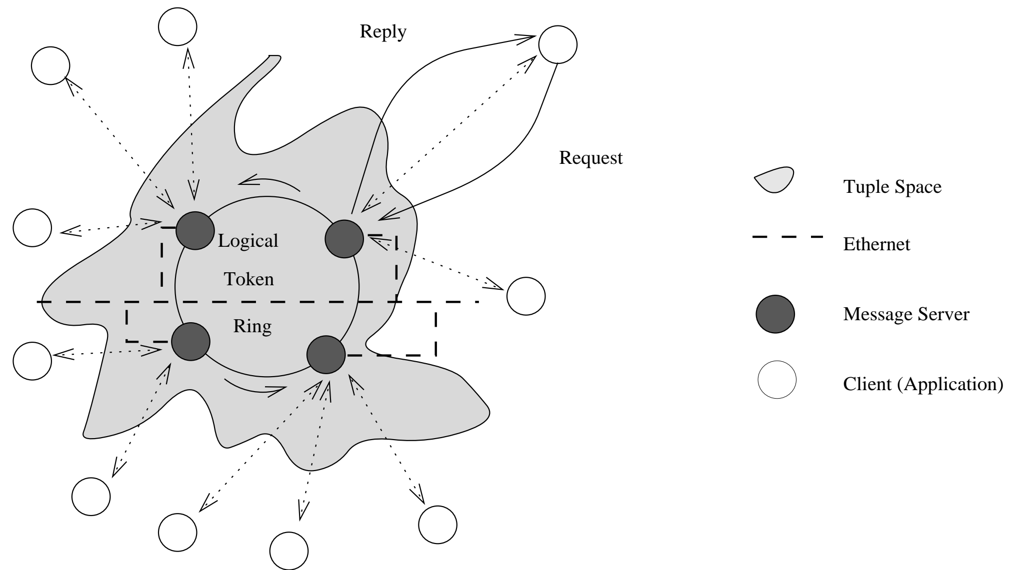

Figure6. Processes of the Tuple Space Machine

residing on different 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.[7]

As shown in Figure 6, 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 for 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 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

[7] 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.

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.

9 Summary

This paper addressed the basic design decisions made when building version 2.4 of the LiPS system for implementing fault-tolerant applications in networks of workstations. The system is currently being used at the University of the Saarland at Saarbrücken, Germany and enables programmers to implement distributed applications using the idle-time of networked workstations.

As the application uses the tuple space for inter-process communication, applications are able to adapt smoothly to the workstation environment. The integration of mechanisms to add some level of software fault-tolerance handles failures like the reboot of a machine. Application processes may be recovered very efficiently using a recovery strategy based on resilient processes and resilient data. The advantage of this strategy is that application processes are independent both in the choice of when to take a checkpoint and when to recover from a checkpoint. This enables exhaustive usage of idle-time present in a workstation network as processes may be migrated to other idle machines in the event the processor they are running on becomes busy. The migration of a process can be based on the mechanisms used to guarantee fault-tolerance. This enables the system to rapidly and easily adapt to changes in machine usability such as those occurring during the daytime.

The above design buys efficiency from the implementation of a Fault-Tolerant Tuple Space Machine, replicating the content of the tuple space among different machines. The LiPS system distinguishes between two runtime systems both based on the Fault-Tolerant Tuple Space Machine. The first runtime system, the so-called system runtime system, provides the applications with software fault-tolerance of level 1 based on a watchdog mechanism. The second runtime system, the so-called application runtime system, provides the application with software fault-tolerance of level 3 based on checkpointing and message logging. The advantage of this strategy

is that application processes are independent in taking a checkpoint. In particular, there is no need to do any synchronization with other application processes when generating a checkpoint.

References

[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.

[Fis96] Fischer J. *Software Fehlertoleranz vom Level 1 in* LiPS. Diplomarbeit, Universität des Saarlandes, 1996. Fachbereich Informatik, Lehrstuhl Prof. Buchmann.

[GCCC85] Gelernter D., Carriero N., Chang S., and Chandran S. Parallel Programming in Linda. *IEEE Transactions on Computer*, 1985.

[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.

[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.

[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.

[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.

[MN94] Mukesh Singhal and Niranjan Shivaratri. *Advanced Concepts in Operating Systems*. Series in Computer Science. Mc Graw Hill, 1994.

[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.

[SF96] Setz T. and Fischer J. Software Fehlertoleranz vom Level Eins in LiPS. In Clemens H. Cap, editor, *Proceedings of SIWORK'96, Workstations and their applications*, pages 102–112, Universität Zürich, Institut für Informatik, May 1996. vdf Hochschulverlag AG an der ETH Zürich.

[SL97] Setz T. and Liefke T. The LiPS Runtime Systems based on Fault-Tolerant Tuple Space Machines. In *Proceedings of the Workshop on Runtime Systems for Parallel Programming (RTSPP), 11th International Parallel Processing Symposium (IPPS'97), Geneva, Switzerland*, April 1997. Appeared as Technical Report, Vrije Universiteit Amsterdam, Faculteit der Wiskunde en Informatica, No. IR-417, februari 1997.

[SR92] Setz T. and Roth R. LiPS: a System for Distributed Processing on Workstations. Technical Report SFB 124 TP D5, Universität des Saarlandes, December 1992.

[SR93] Setz T. and Roth R. Distributed Processing with LIPS. In *ALCOM*, Saarbrücken, August 1993.

[SS83] Schlichting R.D. and Schneider F.B. Fail Stop Processors: An Approach to Designing Fault Tolerant Computing Systems. *ACM Transactions on Computing Systems*, 1(3):222–238, 3 1983.

[ST96] Setz T. and Tews M. Heterogenes checkpointing in LiPS. In Clemens H. Cap, editor, *Proceedings of SIWORK'96, Workstations and their applications*, pages 85–89, Universität Zürich, Institut für Informatik, May 1996. vdf Hochschulverlag AG an der ETH Zürich.

[ST97] Setz T. Integration von Softwarefehlertoleranz in mit LiPS verteilten Anwendungen. In D. Tavangarian, editor, *Proceedings ARCS97, Architektur von Rechensystemen, 14. ITG/GI-Fachtagung, Rostock, Germany*, pages 231–241, Universität Rostock, 9 97. VDE-Verlag.

[STea94] Setz T., Tews M., and et al. *The* LiPS *Development System*, 10 1994. Universität des Saarlandes, Fachbereich Informatik, Lehrstuhl Prof. Buchmann.

[Web95] 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.

[YC83] Yennun H. and Chandra K. Software Implemented Fault Tolerance: Technologies and Experiences. In *Proc. of 23rd IEEE Conference on Fault Tolerant Computing Systems (FTCS)*, pages 2–9, 1983.