Introducing Fault Tolerance to Distributed Maple*

Wolfgang Schreiner   Gabor Kusper
Karoly Bosa
FirstName.LastName@risc.uni-linz.ac.at

Research Institute for Symbolic Computation (RISC-Linz)
Johannes Kepler University, Linz, Austria
http://www.risc.uni-linz.ac.at

January 16, 2001

Abstract

We have extended the parallel computer algebra environment Distributed Maple by fault tolerance mechanisms such that the time spent in a long running computation is not any more wasted by the eventual occurrence of a session failure. The first mechanism is the logging of task return values and of shared object values such that after a failure the newly started session can (transparently to the application program) reuse already computed results. This is complicated by the fact that task arguments and results may embed task handles and that the scheduling layer of Distributed Maple has only a limited amount of information about the activities of the computing layer. The second mechanism is the migration of tasks such that a session may tolerate the failure of individual nodes without overall failure. Both fault tolerance mechanisms are considerably facilitated by the mostly functional nature of the parallel programming model; they allow to run computations that take much longer than the meantime between session failures.

*Supported by grant SFB F013/F1304 of the Austrian Science Foundation (FWF).

1 Introduction

Distributed Maple is a portable and easy to use system for implementing parallel computer algebra algorithms [20]. The software consists of two layers: the *scheduling* layer implemented in Java coordinates the interactions of the *computing* layer which comprises a set of computing engines (e.g. kernels of the computer algebra system Maple). We have employed this system successfully for the parallelization of various methods and applications in algebraic geometry [23, 22, 24]. These programs have been executed in numerous parallel and networked environments, e.g. clusters of Linux PCs and Unix workstations, a Sun HPC 6500 bus-based shared memory machine, an Origin 2000 NUMA multiprocessor, and heterogeneous combinations of these. Recently we have analyzed the system performance in these environments in large detail [21].

The only mechanism originally available in Distributed Maple for handling faults is a watchdog mechanism on every node that regularly checks the connection to the root node; if the root cannot be contacted any more (thus indicating a session failure), the node shuts itself down (thus preventing left over processes). However, as we began to attack larger and larger problems whose solution required more and more time, the meantime between session failures (less than a day) became a limiting factor in the applicability of the system.

There are numerous possibilities for faults that may cause a session failure: a machine becomes unreachable (usually a transient fault, i.e., the machine is rebooted or is temporarily disconnected), a process in the scheduling layer or in the computation layer crashes for mysterious reasons (a bug in the implementation of the Java Virtual Machine, of the Java Development Kit, of Maple, or possibly also of the Distributed Maple software) or the computation itself aborts unexpectedly (a bug in the application). While the last issue can be eventually overcome and the Distributed Maple software itself is very stable, there certainly exist bugs in the lower software levels that are partially out of our control; machine/network/operating system faults may happen in any case.

We have therefore started to investigate how to introduce fault tolerance mechanisms into Distributed Maple that let the system deal with faults in such a way that the time spent in long running session is not wasted by an eventual failure. There exist various fundamental mechanisms to achieve fault tolerance in distributed systems [12, 4]:

- We may organize a distributed activity as a collection of *transactions* whose effects become durable on commit time; this is a popular basis for database-oriented applications but does not integrate well with parallel programming models.
- By *replication* we understand the duplication of resources or services on multiple servers such that, if a server fails, a request is handled by another one; because parallel programming systems are concerned about optimal use of resources, only few apply this approach [15, 1].
- Most frequently, parallel programming environments pursue fault tolerance by *checkpointing*: they regularly save global snapshots of the session

state on stable storage such that a failed session can be re-started from the last saved state (potentially migrating tasks to other machines); stable storage can be implemented by disks or by replicated copies in the memories of multiple machines [17, 19].

Some systems perform checkpointing transparently to the application, often on top of PVM [6, 5, 9, 13] or MPI [18]. Other systems rely on application support for checkpointing [3, 16].

- Most metacomputing and message passing frameworks include at least *failure detection and handling* services that enable the user to deal with failures entirely on the application-level, e.g. the Globus toolkit [25], the Harness environment [14], PVM [8] or FT-MPI [7].

All these approaches are relatively complex because they have to deal with general parallel computations; for special problems much simpler solutions exist [10]. However, also parallel programming models that are more abstract than message passing should allow to deal with fault tolerance in a simpler way. While this is not yet completely true for the coordination language Linda [2], the *functional* programming model provides this potential to a large degree [11].

Distributed Maple runs programs in the imperative language of Maple, but its *parallel* programming model is essentially functional: it provides the ability to spawn function applications as concurrent tasks and to wait for their results (extended by a non-deterministic synchronization facility and single assignment shared data objects). Still the model is very general, e.g., it allows tasks to start other tasks, to receive the handles of other tasks as arguments and return task handles as results (and thus supports general communication structures).

Our primary goal is to extend the scheduling layer of Distributed Maple to provide fault-tolerance in a way that is *transparent* to the application. However, computation layer and scheduling layer are clearly separated by a communication protocol such that the scheduling layer is completely unaware of the actual nature of the computation. Our secondary goal is to preserve this distinction such that the use of the scheduling layer for different kinds of computing engines is not compromised. We thus have to deal with the *limited information* that the scheduling layer has about the computation.

It is the main contribution of this paper to show how these two goals can be achieved with comparatively simple means by profiting from the functional nature of the parallel programming model. Because of the limited scalability of many computer algebra application, arbitrary scalability of the system and avoiding single point of failures has not yet been an issue.

The remainder of this paper is organized as follows: Section 2 introduces the model of the system and its operation on which the following descriptions are based. Sections 3, 4, and 5 explain in increasing order of generalization the logging and failure recovery mechanisms that we have introduced to reuse already computed results. Section 6 extends this by a mechanism that allows the system to cope with node failures without overall failure. Section 7 summarizes our results and outlines further development directions.

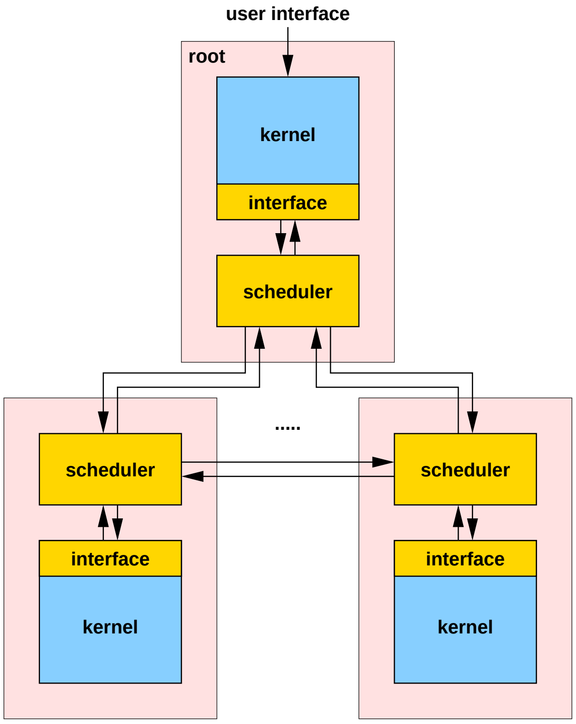

Figure 1: System Model

2 System and Execution Model

A Distributed Maple session comprises a set of *nodes* (machines, processors) each of which holds a pair of processes: a (e.g. Maple) *kernel* which performs the actual computation, and a *scheduler* (a Java Virtual Machine executing a Java program) which distributes and coordinates the tasks created by all kernels (see Figure 1). The scheduler communicates, via pairs of sockets, with the kernel on the same node and, via sockets, with the schedulers on the other nodes. The *root* is that node from which the session was established by user interaction with the kernel. Initially, a single task is executing on the root kernel; this task may subsequently create new tasks which are distributed via the schedulers to other kernels and may in turn create new tasks.

The parallel programming model is basically functional (see Figure 2): from the perspective of a scheduler, a kernel may emit a number of messages $\mathsf{task}{:}\langle t, d\rangle$ where t is the identifier of the task and d represents its description. Such a task needs to be eventually forwarded to some idle kernel which eventually returns a message $\mathsf{result}{:}\langle t, r\rangle$ where r represents the computed value. When a kernel emits on behalf of the currently executing task a message $\mathsf{wait}{:}\langle t\rangle$, this task is assumed to be blocked until the scheduler responds with the corresponding result. Thus, if this result not yet available, the scheduler may submit to the now idle kernel another task for execution; when this task has returned its result, the kernel may receive the result waited for or yet another task.

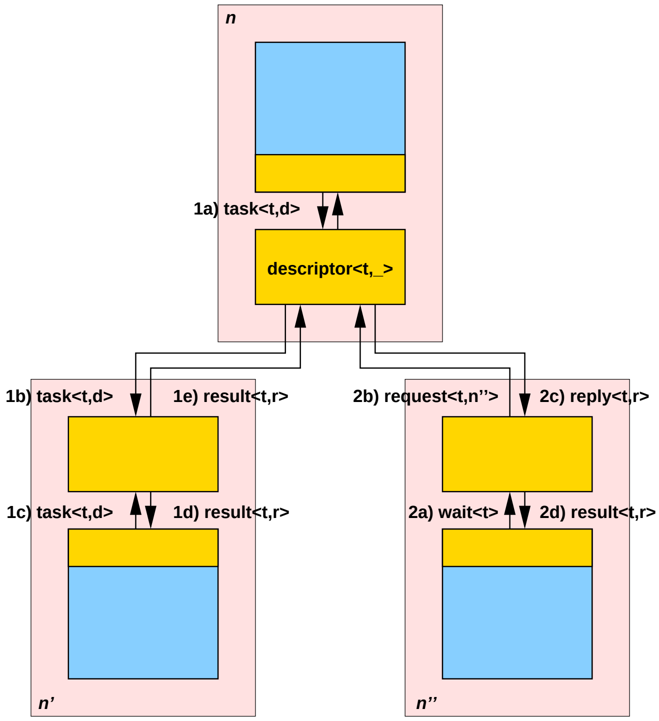

Figure 2: Execution Model

A task identifier t is a word that encodes a pair $\langle n, i \rangle$ where n is a node index and i is a task index. Initially, every kernel receives from the scheduler the index n of the current node and an interval $[i_{\min}, i_{\max}]$ which it may use to assign a unique identifier $t = \langle n, i \rangle$ with $i_{\min} \leq i \leq i_{\max}$ to every newly created task (the scheduler reserves part of the representable index range for the creation of additional kernels). The identifier t of a task thus always describes the node n on which the task was created; this node thus serves as the rendezvous point between node n' computing the result of t and node n'' requesting this result.

When a scheduler on node n receives a task:$\langle t, d \rangle$ from its kernel, it thus allocates a result descriptor that will eventually hold the corresponding value r; the task itself is scheduled for execution on some node n'. When a kernel on some node n'' issues a wait:$\langle t \rangle$, the scheduler on n'' forwards a request:$\langle t, n'' \rangle$ to n. If r is not yet available on n, this request is queued in the result descriptor. When the kernel on n' eventually returns the result:$\langle t, r \rangle$, the scheduler on n' forwards this message to n, which constructs a reply:$\langle t, r \rangle$ and sends it to n''.

3 Logging Results: First-Order Tasks

Our first step towards fault tolerance is to log task results on stable storage. If the session fails for any reason, we can thus re-run the session and read the logged results without re-executing the corresponding tasks. Our only assumption is

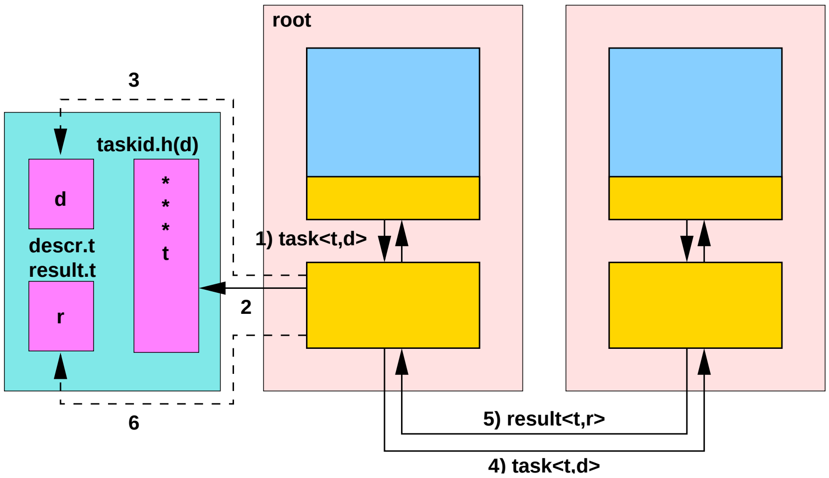

Figure 3: Logging Results: Fist-Order Tasks

that the root has access to a writable file system; this file system implements the stable storage and the root becomes in charge of maintaining the log.

For the moment, we restrict our consideration to programs whose tasks are *first order*: neither any task description d nor any task result r contains some task identifier t, i.e., task identifiers are not passed as parts of task arguments and are not returned as parts of task results.

The logging mechanism utilizes the fact that in Distributed Maple the root is in charge of task distribution: every new task:$\langle t, d\rangle$ is forwarded to the root which eventually assigns it to some node for execution. The operations dealing with result logging and failure recovery are as follows (see Figure 3):

Logging When the root receives a task:$\langle t, d\rangle$, it computes a numeric hash code $h(d)$ and appends to file `taskid.`$h(d)$ the task identifier t. Then the root starts an asynchronous thread to write the task description d into a new file `descr.`t.

When a node sends a result:$\langle t, r\rangle$ to some node n different from the root, it forwards a copy to the root. When the root receives this result, it creates an asynchronous thread to write the task result r into a new file `result.`t.

All data are written in a format that enables a reader to recognize incomplete writes. At any time, `taskid.`$h(d)$ holds a sequence of task identifiers t (the last of which may be incomplete) for which there may exist description files `descr.`t and/or result files `result.`t (both with possibly incomplete contents). When the session terminates without failure, the log files are discarded.

Recovery When a session is re-started after a failure, the root may receive from a node n a task:$\langle t, d\rangle$ such that a file `taskid.`$h(d)$ exists. If for some complete task identifier t' in this file there exist file `descr.`t' with a complete description identical to d and file `result.`t' with a complete result r, the root need not schedule this task for execution but may immediately return result:$\langle t, r\rangle$ to n.

The comparison of task descriptions is required because task identifiers need not be identical across sessions; since the result r of a task only depends on its description d, the identity of descriptions is a sufficient condition to ensure the identity of results.

The mechanism is simple and efficient: the only overhead occurs on the root for writing the log files (the potentially large task descriptions and results are written by asynchronous threads) and, if a task is created on a node other than the root, the sending of a duplicate result to the root.

4 Logging Results: Higher-Order Tasks

The mechanism described in the previous section guarantees correct results if task arguments and results do not embed task identifiers. However, assume that a task t creates another task and embeds its identifier t' in result r. If r is logged and the session fails, in the recovery session this result may be read from the log such that task identifier t' is re-created. A task may subsequently issue a wait:$\langle t'\rangle$ referring to a no more existing task or, even worse, to a task that computes a different result than in the failed session.

In order to allow *higher-order tasks* (tasks which receive task identifiers as arguments or return them as results), we introduce a *session identifier* s which may be used to distinguish task identifiers from different sessions. In a session that does not represent the recovery of a previously failed session, the session identifier is initialized to 0 and a file `session` is written with content 0. In a recovery session, the previous identifier s is read from `session`, the new session identifier is taken as $s+1$ and overwrites the content of `session`. If the recovery session also fails, a new recovery session may be initiated and thus the session identifier may grow to an arbitrary size (subject to an implementation limit).

A *task identifier* now encodes a triple $\langle s, n, i\rangle$ that embeds the number s of the session in which the corresponding task was created. The mechanism described in the previous section is now generalized as follows:

Logging When a kernel on a node different from the root creates a task, the system operates according to one of two modes that can be selected on the startup of the session:

Fast Mode The systems process as described in the previous section. This mode does not impose additional overhead, but it may be that after a crash a recovery session cannot proceed and the whole session has to be re-executed from scratch (see below).

Safe Mode The scheduler forwards the task message to the root and, until it receives an acknowledgement message from the root, does *not* process any further messages from the kernel. On receipt of this message, the root creates an entry in the corresponding hash file, writes

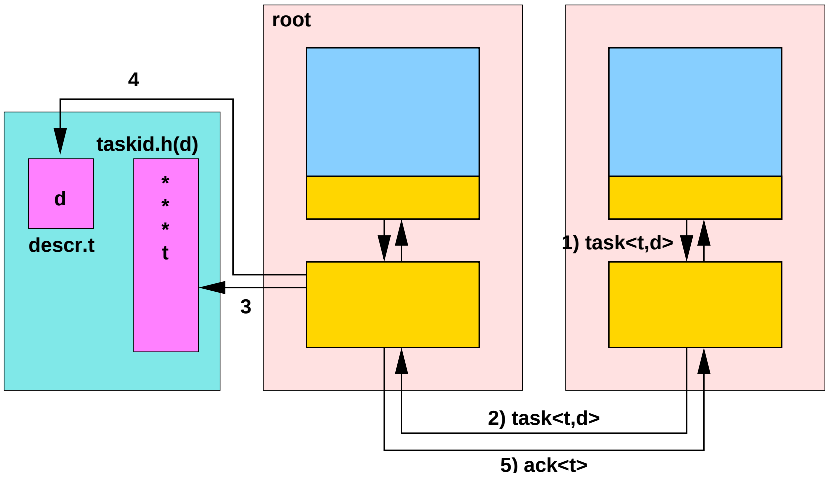

Figure 4: Logging Results: Higher-Order Tasks (Safe Mode)

synchronously the description and then returns the acknowledgement which enables the scheduler to process further kernel messages (see Figure 4).

A task t that has created another task t' is thus not able to disseminate the identifier t' before the description of t' is appropriately logged on the root. Consequently, if a logged task result contains a reference to t', it is guaranteed that the root holds the description of t' in the log. This mode therefore may cause a kernel to idle more than in the fast mode, but it makes sure that a potential recovery session can proceed as planned.

Recovery In addition to the actions described in the previous section, we have to deal with task identifiers that may (via logged descriptions or results) refer to previous sessions: the root is in charge of such tasks.

If a kernel on node n issues a wait:$\langle t \rangle$ where the session identifier of t is not that of the current session, the scheduler on n sends a request:$\langle r, n \rangle$ to the root. If the root receives this request, it looks up whether it holds a result descriptor for t; if yes, it proceeds as usual, i.e., it responds with the result or, if this is not yet available, queues the request in the descriptor.

If the root does not hold a result descriptor for t, it creates one and queues the request there. It then looks up file `result.`t for the result of t logged in a previous session. If this file exists and holds a complete result r, the scheduler writes r into the descriptor and responds with reply:$\langle t, r \rangle$.

Otherwise, the scheduler looks up the file `descr.`t for a description d of t. If the file does not exist or holds an incomplete description (which is only possible if the task t was created in a session operating in fast mode), the scheduler aborts the session; the computation has then to be restarted from scratch.

Otherwise, the scheduler creates a new task:$\langle t, d \rangle$ which is handled as usual. When a kernel issues a $\mathsf{result}{:}\langle t, r\rangle$ of a task created in a previous session, the scheduler forwards this result to the root.

Above logging and recovery mechanism (in safe mode) guarantees that a recovery session s can proceed to complete the execution of a previously failed session making use of all results logged in any session $s' < s$.

5 Logging Results: Shared Objects

The functional model is not the only one by which tasks in Distributed Maple may interact with each other: they may also use self-synchronized *shared objects* for a producer/consumer (or, more general, dataflow-like) kind of operation. From the perspective of the scheduler, a kernel may issue a message $\mathsf{data}{:}\langle t\rangle$ which requests the allocation of a shared object with identifier t taken from the range of task identifiers assigned to the kernel. Actually, a shared object is nothing but a result descriptor described in Section 2; the only difference is that it receives, *if it is still empty*, its value r by a message $\mathsf{put}{:}\langle t, r\rangle$ that may be issued by any task knowing t.

The logging and recovery mechanism is extended as follows:

Logging If a $\mathsf{data}{:}\langle t\rangle$ message is issued, the system operates analogously to a $\mathsf{task}{:}\langle t, d\rangle$ call as described in the previous section (with the two modes "fast" and "safe").

However, since there is no task description d, no hash file is written and no description file is created; instead a file `data.`t is (synchronously) written to indicate that identifier t refers to a shared object. If a $\mathsf{put}{:}\langle t, r\rangle$ is issued on a node different from the root, a duplicate of this message is forwarded to the root and processed analogously to $\mathsf{result}{:}\langle t, r\rangle$ messages for task results by writing `result.`t.

Recovery If the root receives a $\mathsf{wait}{:}\langle t\rangle$ message where t refers to a previous session and is destined for a shared object (as indicated by the existence of file `data.`t), it proceeds analogously to the description in the previous section except for the following situation: if the result descriptor does not yet exist, the root creates one and records the request. If the complete result r of t is logged, the request may be immediately answered by a $\mathsf{result}{:}\langle t, r\rangle$. Otherwise, the root need not do anything because the result will be computed by another task t'.

We can argue for the correctness of the last statement as follows: let t' be the task in charge of delivering r (if no such task exists, r is never computed, anyway). Since r is not yet logged, also the result of t' is not yet logged (because all messages are transmitted and processed in FIFO order). Now let T be the sequence of all $\mathsf{wait}{:}\langle * \rangle$ requests that would be performed if every task would be executed only on demand of its result. If this sequence contains a request for the result of t' before any request for r, then the recovery mechanism eventually triggers (as shown in the previous section) the execution of t' and the delivery of r.

As the previous paragraph shows, programs that operate with shared data need to satisfy some constraint to profit from the logging and recovery mechanism; if violated, the recovery session may run into a deadlock. In a nutshell, the constraint says that the initial task eventually needs to wait (directly or indirectly) for the termination of every task that fills a shared object and it needs to wait on it before it (directly or indirectly) depends on the availability of the object content. Fortunately, many real programs can be easily checked and, if necessary, augmented and restructured to satisfy this constraint.

As another restriction, the logging and recovery mechanism cannot deal with shared objects that are cleared and re-filled [20].

6 Tolerating Node Failures

The previous sections dealt with mechanisms to ensure that, when a fault causes the failure of a session, a subsequent recovery session may reuse previously computed results. In this section, we sketch a mechanism that enables a session to cope with faults *without* aborting. We restrict our attention to the following scenario: a machine executing a non-root node becomes unreachable (stop failure) and the root continues operation with the remaining nodes (if the root fails, the session also fails).

A necessary condition to detect this kind of failure is that the root cannot contact some node for a certain period of time. We thus let the root periodically check whether a message has been recently received from every node and, if not, send a `ping` message that has to be acknowledged. If no acknowledgement arrives within a certain time bound, this node is considered as *dead*. However, this does not necessarily mean that the node is actually dead; it may be slow in responding, the connection to the root may have been transiently interrupted, or the connection to the root is permanently broken but connections to other nodes still exist. We therefore must assume that even a dead node still may send messages to the root or to any other node. Thus, when the root designates a node as dead, it informs all other nodes correspondingly: every node closes the connection to the dead node and ignores any remaining messages from this node (such messages may arrive between the handling of the notification and the closing of the connection).

There are two main problem that the root now has to deal with:

1. the management of all result descriptors that have been stored on the dead node, and
2. the rescheduling of all tasks that were executing on the node at the time of its alleged death.

Since the root is in charge of task scheduling, the root sees every task created in the session. Furthermore, by the logging mechanism discussed in the previous sections, the root sees every result computed in the session. For every node n, the root can therefore maintain two sets T_n and S_n:

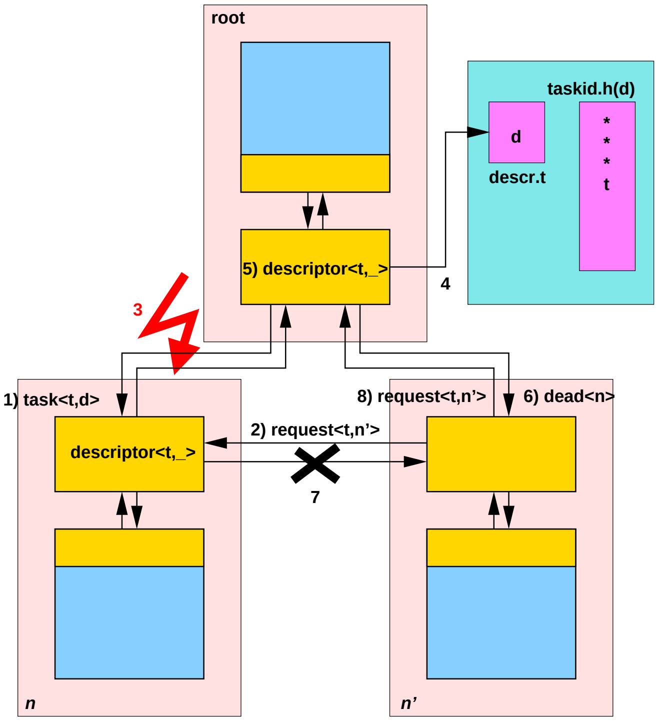

Figure 5: Tolerating Node Failures

1. T_n denotes all tasks scheduled on n; for a subset T_n^{r} the results are available (in the logging files). All tasks in $T_n - T_n^{\mathrm{r}}$ have to be executed again; the root puts them back into the pool of tasks to be scheduled for execution.

2. S_n denotes all tasks or shared objects whose descriptors are stored on n; for a subset S_n^{r} the results are available (in the logging files). The root becomes the owner of elements in S_n; it allocates the corresponding result descriptors and, for all elements of S_n^{r}, fills them with results.

   Subsequently, every node will send requests for a result in S_n to the root. However, there may be still outstanding requests sent to n but not yet answered at the time of its death. Every node n' therefore holds a table R_n of all request:$\langle t, n' \rangle$ messages sent to node n but not yet answered by a reply:$\langle t, r \rangle$. When n is marked dead, the node re-sends all messages in R_n to the root which will eventually answer them (see Figure 5).

Thus all tasks scheduled on an eventually dead node n are executed (possibly on a different node n') and every descriptor originally housed by n finds a new home on the root to which all open and all future requests are redirected.

7 Conclusions

We have implemented in Distributed Maple the logging and recovery mechanisms described in Sections 3, 4, and 5 such that all program sessions operating in the functional parallel programming model can be recovered; program sessions using shared objects can be recovered provided that the program satisfies certain constraints. An implementation of the task migration mechanism described in Section 6 is under way such that sessions can continue operation even if nodes become unreachable. Up to now, the problem sizes that Distributed Maple could successfully deal with was limited by the meantime between failures which was less than a day or so. With the new fault tolerance mechanisms, we will be able to tackle problems whose solution takes much longer.

Our work shows how distributed computations that operate with an essentially functional parallel programming model can tolerate faults with relatively simple mechanisms, i.e., without global snapshots as required in message passing programs. The runtime overhead imposed by the logging mechanism is very moderate; adding tolerance of node failures on top does then not require much extra overhead. One reason for this simplicity is the delegation of all logging activities to a single root node that also performs the task scheduling decisions (and remains a single point of failure); the model is therefore not scalable beyond a certain number of nodes. However, it is suitable for the system environments that we have used up to now ($\leq$ 24 nodes); many parallel computer algebra applications do not scale well enough to profit from considerably more nodes. Nevertheless, improving the scalability of Distributed Maple and avoiding the single point of failure remains an open issue.

References

[1] David M. Arnow. DP: A Library for Building Portable, Reliable Distributed Applications. In *1995 USENIX Technical Conference*, pages 235–247, New Orleans, Louisiana, January 16–20, 1995. USENIX.

[2] David E. Bakken and Richard D. Schlichting. Supporting Fault-Tolerant Parallel Programming in Linda. *IEEE Transactions on Parallel and Distributed Systems*, 6(3):287–302, March 1995.

[3] Adam Beguelin, Erik Seligman, and Peter Stephan. Application Level Fault Tolerance in Heterogeneous Networks of Workstations. *Journal of Parallel and Distributed Computing*, 43(2):147–155, June 1997.

[4] Kenneth P. Birman. *Building Secure and Reliable Network Applications*. Manning, Greenwich, Conneticut, 1996.

[5] Jeremy Casas, Dan Clark, Phil Galbiati, Ravi Konuru, Steve Otto, Robert Prouty, and Jonathan Walpole. MIST: PVM with Transparent Migration and Checkpointing. In *Third Annual PVM User's Group Meeting*, Pittsburgh, Pennsylvania, May 1995.

[6] A. Clematis and V. Gianuzzi. CPVM — Extending PVM for Consistent Checkpointing. In *4th Euromicro Workshop on Parallel and Distributed*

Processing (PDP'96), pages 67–76, Braga, Portugal, January 24–26, 1996. IEEE Computer Society Press.

[7] Graham E. Fagg and Jack J. Dongarra. FT-MPI: Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, Proceedings of the 7th European PVM/MPI Users' Group Meeting*, volume 1908 of *Lecture Notes in Computer Science*, pages 346–353, Balatonfred, Hungary, September 10–13, 2000. Springer.

[8] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine — A Users' Guide and Tutorial for Networked Parallel Computing.* MIT Press, Cambridge, Massachusetts, 1994.

[9] V. Gianuzzi and F. Merani. Using PVM to Implement a Distributed Dependable Simulation System. In *3rd Euromicro Workshop on Parallel and Distributed Processing (PDP'95)*, pages 529–535, San Remo, Italy, January 25–27, 1995. IEEE Computer Society Press.

[10] Adriana Iamnitchi and Ian Foster. A Problem Specific Fault Tolerance Mechanism for Asynchronous, Distributed Systems. In *International Conference on Parallel Processing (29th ICPP'00)*, Toronto, Canada, August 21–24, 2000. Ohio State University.

[11] R. Jagannathan and E. A. Ashcroft. Fault Tolerance in Parallel Implementations of Functional Languages. In *21st International Symposium on Fault-Tolerant Computing*, pages 256–263, Montreal, Canada, June, June 1991. IEEE Computer Society Press.

[12] Pankaj Jalote. *Fault-Tolerance in Distributed Systems.* Prentice Hall, Englewood Cliffs, NJ, 1994.

[13] Juan Leon, Allan L. Fisher, and Peter Alfons Steenkiste. Fail-safe PVM: a Portable Package for Distributed Programming with Transparent Recovery. Technical Report CMU-CS-93-124, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, February 1993.

[14] M. Migliardi, V. Sunderam, A. Geist, and J. Dongarra. Dynamic Reconfiguration and Virtual Machine Management in the HarnessMetacomputing System. In *Computing in Object-Oriented Parallel Environments — Second International Symposium (ISCOPE 98)*, volume 1505 of *Lecture Notes in Computer Science*, pages 127–134, Santa Fe, New Mexiko, December 8–11, 1998. Springer.

[15] Özalp Babaoglu, Lorenzo Alvisi, Alessandro Amoroso, Renzo Davoli and Luigi Alberto Giachini. Paralex: An Environment for Parallel Programming in Distributed Systems. In *1992 International Conference on Supercomputing*, pages 187–187, Washington, DC, July 19–24, 1992. ACM Press.

[16] Taesoon Park and Heon Y. Yeom. Application Controlled Checkpointing Coordination for Fault-tolerant Distributed Computing Systems. *Parallel Computing*, 26(4):467–482, March 2000.

[17] James S. Plank, Youngbae Kim, and Jack Dongarra. Algorithm-Based Diskless Checkpointing for Fault Tolerant Matrix Operations. In *25th International Symposium on Fault-Tolerant Computing*, Pasadena, California, June 1995. IEEE Computer Society Press.

[18] Samuel H. Russ, Jonathan Robinson, Brian K. Flachs, and Bjorn Heckel. The Hector Distributed Run-Time Environment. *IEEE Transactions on Parallel and Distributed Systems*, 9(11):1104–1112, November 1998.

[19] Fred B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[20] Wolfgang Schreiner. Distributed Maple — User and Reference Manual. Technical Report 98-05, Research Institute for Symbolic Computation (RISC-Linz), Johannes Kepler University, Linz, Austria, May 1998. http://www.risc.uni-linz.ac.at/software/distmaple.

[21] Wolfgang Schreiner. Analyzing the Performance of Distributed Maple. Technical Report 00-32, Research Institute for Symbolic Computation (RISC-Linz), Johannes Kepler University, Linz, Austria, November 2000.

[22] Wolfgang Schreiner, Christian Mittermaier, and Franz Winkler. Analyzing Algebraic Curves by Cluster Computing. In Peter Kacsuk and Gabriele Kotsis, editors, *Distributed and Parallel Systems – From Instruction Parallelism to Cluster Computing, Proceedings of DAPSYS'2000, 3rd Austrian-Hungarian Workshop on Distributed and Parallel Systems*, pages 175–184, Balatonf¡FC¿red, Lake Balaton, Hungary, September 10–13, 2000. Kluwer Academic Publishers, Boston.

[23] Wolfgang Schreiner, Christian Mittermaier, and Franz Winkler. On Solving a Problem in Algebraic Geometry by Cluster Computing. In Arndt Bode, Thomas Ludwig, Wolfgang Karl, and Roland Wismüller, editors, *Euro-Par 2000, 6th International Euro-Par Conference*, volume 1900 of *Lecture Notes in Computer Science*, pages 1196–1200, Munich, Germany, August 29 - September 1, 2000. Springer, Berlin.

[24] Wolfgang Schreiner, Christian Mittermaier, and Franz Winkler. Plotting Algebraic Space Curves by Cluster Computing. In Xiao-Shan Gao and Dongming Wang, editors, *ASCM'2000, 4th Asian Symposium on Computer Mathematics*, pages 49–58, Chiang Mai, Thailand, December 17-21, 2000. World Scientific Publishers, Singapore/River Edge.

[25] P. Stelling, C. Lee, I. Foster, G. von Laszewski, and C. Kesselman. A Fault Detection Service for Wide Area Distributed Computations. In *Seventh IEEE International Symposium on High Performance Distributed Computing*, July 28–31, Chicago, Illinois, 1998. IEEE Computer Society Press.