TOLERATING STOP FAILURES IN DISTRIBUTED MAPLE*

KÁROLY BÓSA† AND WOLFGANG SCHREINER†

Abstract. In previous work we have introduced some fault tolerance mechanisms to the parallel computer algebra system Distributed Maple such that a session may tolerate the failure of computing nodes and of connections between nodes without overall failure. In this paper, we extend this fault tolerance by some advanced mechanisms. The first one is the reconnection of a node after a connection failure such that a session does not deadlock. The second mechanism is the restarting of a node after a failure such that the session does not fail. The third mechanism is the change of the root node such that a session may tolerate also the failure of the root without overall failure.

Key words. distributed systems, fault tolerance, computer algebra.

1. Introduction. Distributed Maple (see Figure 1.1) is a Java-based system for implementing parallel programs in the computer algebra system Maple [26]. We have employed this system successfully for the parallelization of various methods and applications in algebraic geometry [28, 29]. The system is portable and has been used 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 [27].

We have used the system to develop the parallel versions for various non-trivial applications from computer algebra and algebraic geometry, e.g. bivariate resultant computation, real root isolation, plotting of algebraic plane curves, plotting of surface to surface intersections and neighborhood analysis of algebraic curves. A comprehensive survey on the system and its applications is given in [31].

Distributed Maple has evolved from our own experience in the development of parallel computer algebra environments and from learning from the work of other researchers. The programming interface of the system is based on a para-functional model as adopted by the PARSAC-2 system [16] for computer algebra on a shared memory multiprocessor. The model was refined in PACLIB [11] on which a para-functional language was based [25].

The only mechanism originally available in Distributed Maple for dealing with faults was the watchdog mechanism for shutting down the system in case of failures. However, as we began to attack larger and larger problems, 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 (a bug in the implementation of the Java Virtual Machine, of the Java Development Kit, of Maple, or of Distributed Maple itself) or the computation itself aborts unexpectedly (a bug in the application). While the last issue can be overcome and the Distributed Maple software itself is very stable, there certainly exist bugs in the lower software levels that are out of our control; machine/network/operating

*Sponsored by Upper Austrian Government (Ph.D. scholarship) and SFB/FWF project P1302.

†*Research Institute for Symbolic Computation (RISC-Linz), Johannes Kepler University, Linz, Austria*, http://www.risc.uni-linz.ac.at, e-mail: `FirstName.LastName@risc.uni-linz.ac.at`.

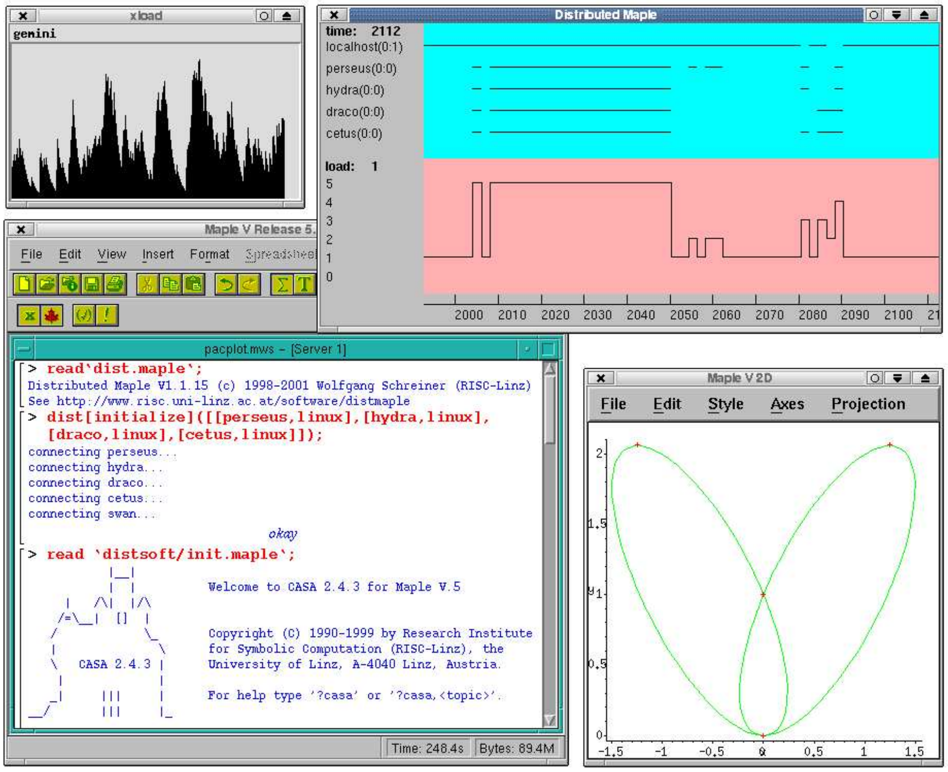

FIG. 1.1. *The User Interface of Distributed Maple*

system faults may happen in any case.

We have therefore started to investigate how to introduce fault tolerance mechanisms 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 [5, 13]:

(i) 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 systems apply this approach [1, 2].

(ii) 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 [22, 24]. Some systems perform checkpointing transparently to the application, often on top of PVM [7, 6, 10, 17, 15] or MPI [23]. Other systems rely on application support for checkpointing [4, 21].

(iii) Many metacomputing and message passing frameworks do not provide built-in fault tolerance at all. Rather they include failure detection and handling services that enable the user to deal with failures at the application-level. Examples of such systems are the Globus toolkit [32], the Harness environment [18], PVM [9] or FT-MPI [8].

All these approaches are relatively complex because they have to deal with general parallel computations; for special problems much simpler solutions exist [12]. 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 [3], the *functional* programming model provides this potential to a large degree [14].

All of the possibly fault cases mentioned above can be classified into the following failure types: *Message* failures, *Stop* failures or *Byzantine* failures [19]. We do not deal with tolerating of *Message* failures and *Byzantine* failures (our system currently is not able to handle corrupt messages and corrupt task executions), but in the functional parallel programming model, the handling of these two kinds of failure situations may happen similarly as in the message passing or any other general parallel programming model (there are no simpler solutions for these in the functional model). Furthermore, there exist already numerous general and effective algorithms for tolerating of these kinds of failures [19].

We focus on *Stop* failures and we assume in this paper that only such failures may occur (a part of the system can exhibit *Stop* failure simply by stopping somewhere in the middle of its execution). In more detail, we deal with the following error situations: the root of the tree of computation nodes crashes, some non-root node crashes, some Maple process fails, a connection between the root and another node breaks and a connection between two non-root nodes breaks. We say that a node crashes if the machine stops execution or reboots, or if the Distributed Maple process aborts.

In the original system, we had to restart the computation in each of these cases. If the root crashes, the system aborts; in all other cases, it deadlocks and must be stopped manually. Our goal was to extend the time scale of distributed computations (measured in hours at the moment) to many days as is required for realistic applications. Another requirement was that the implemented fault tolerance has to be *transparent* to the application.

Earlier, we introduced three fault tolerance mechanisms, by which some of the mentioned *Stop* failure problems were covered. The functional task model of Distributed Maple considerably simplified these developments [30].

Now, we have extended these mechanisms to cover more *Stop* failures: first, a session may tolerate node failures (except root failure) and connection failures without loss of resources (e.g.: computed task results, Maple kernels or schedulers); second, we have introduced a mechanism to avoid a session failure, even if the root becomes unreachable for most of the non-root nodes. With these mechanisms, Distributed Maple is by far the most advanced system for computer algebra concerning reliability in distributed environments.

The remainder of this paper is organized as follows: Section 2 introduces the model of the system and its operation. Section 3 gives a short overview about the previously achieved fault tolerance mechanisms on which the following descriptions are based. Section 4 explains the reconnection and restarting mechanisms that we have introduced to reduce the loss of resources after node or connection failures. Section 5 extends this by a mechanism that allows the system to cope with root node failure without overall failure. Section 6 compares the fault tolerance features of Distributed Maple with a checkpointing mechanism developed for a particular parallel programming environment. Section 7 summarizes our results and outlines further development directions.

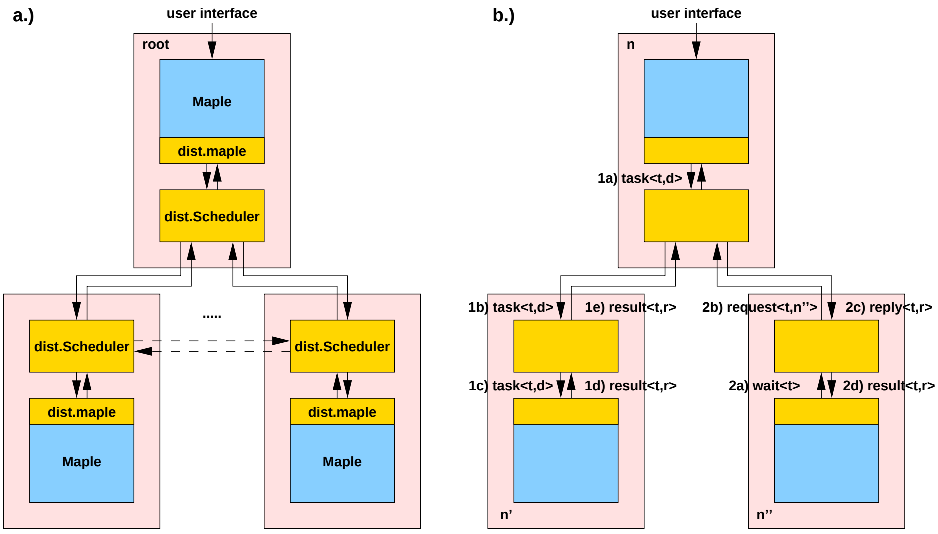

FIG. 2.1. ***a.)*** *System Model* ***b.)*** *Execution Model*

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 2.1a). 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 communication protocol between the scheduler and the kernel is implemented by *dist.maple* interface. 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 2.1b): from the perspective of a scheduler, a kernel may emit a number of messages `task:`$<t, d>$ where t identifies the task and d describes it. The task needs to be forwarded to some idle kernel which eventually returns a message `result:`$<t, r>$ where r represents the computed result. When a kernel emits `wait:`$<t>$, this task is blocked until the scheduler responds with the result of t. Thus, if this result is not yet available, the scheduler may submit to the now idle kernel another task; when this task has returned its result, the kernel may receive the result waited for or yet another task. A task identifier t encodes a pair $<n, i>$ where n identifies the node on which the task was created and i is an index. The node n thus serves as the rendezvous point between node n' computing the result of t and node n'' requesting this result. When the scheduler on n receives a `task:`$<t, d>$ from its kernel, it thus allocates a result descriptor that will eventually hold the result r; the task itself is scheduled for execution on some node n'. When a kernel on some node n'' issues a `wait:`$<t>$, the scheduler on n'' forwards a `request:`$<t, n''>$ 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:`$<t, r>$, the scheduler on n' forwards this message to n, which constructs a `reply:`$<t, r>$ and sends it to n''.

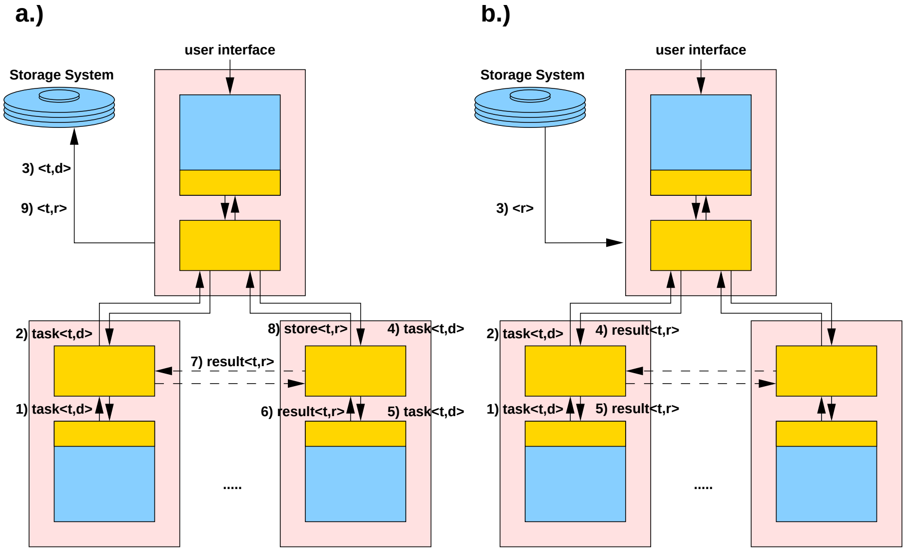

FIG. 3.1. ***a.)*** *Logging a Result* ***b.)*** *Restoring a Result*

3. Formerly Achieved Fault Tolerance. The only mechanism originally available in Distributed Maple for handling faults was a watchdog mechanism on every non-root node that regularly checked whether a message had been recently received from the root. If not, it sent a `ping` message that had to be acknowledged. If no acknowledgement arrived from the root within a certain time bound, the node shut itself down.

We have extended this basic operation by three fault tolerance mechanisms. For the correct working of these mechanisms, we assume that the system has access to a reliable and stable storage system which never loses data. Communication is reliable, i.e. messages are never corrupted or duplicated (no *Message* failures) and they may be lost only in *Stop* failure situations. No *Byzantine* failures may occur.

Logging Results The logging mechanism can store a consistent state of a session and is able to recover this stored state after a failure [30]. This mechanism is based on the scheduling mechanism originally available in Distributed Maple, which allows the root to receive all task descriptions created by all nodes (see Figure 3.1a). Before the root schedules a `task:`$<t,\ d>$, it saves d. When a node sends a `result:`$<t\ ,r>$ to some other node different from the root, it forwards a copy in a `store:`$<t,\ r>$ to the root which saves it. When a session is restarted after a failure (see Figure 3.1b), the root may receive a `task:`$<t',\ d>$. If d is already stored, the root checks whether there exists stored r for the received d and it tries to restore this r. The mechanism is able to recognize the corrupted files and it uses only the properly stored data in the recovery phase.

Tolerating Node Failures The system is capable to cope with node failures without overall failure [30]. This mechanism is based on the original watchdog mechanism. It sends a `ping` to a node and supposes its failure, if it does not receive any reply in a limited time period. If the root observes that a node become unreachable, the root declares it as *dead*. The "Tolerating Node Failures" mechanism is based on the "Logging Results" mechanism, because all results that have been stored on the dead node are recovered and provided by the root. The root also recovers and reschedules to another node those task descriptors that were under processing on the dead node.

Tolerating Peer Connection Failures The connection between two non-root nodes is called peer connection. Such connections are established on demand during the execution. The system is capable to cope with peer connection failures without overall failure. This mechanism is based on "Tolerating Node Failures" mechanism, because it assumes that the system is able to continue normal operation, if a connection between root and another node fails. The principle is simple: if a non-root node cannot send a message to another such node, it sends this message to the root which forwards it to the final destination.

The effect of these mechanisms on the performance of the system has not been measured yet, but we do not expect it to be significant. For instance in the case of the "Logging Results" mechanism, task descriptions and task results are saved by separate threads and thus do not hamper the normal flow of operation. These descriptions and results are also communicated in the basic operation model; their size is therefore bound by the performance of the communication system rather than by the extra overhead of the saving mechanism. Additionally, the "Logging Results" mechanism introduces only one extra message type, the `store:`$<t, r>$ message, for saving of the task results to the storage system (task descriptions are saved when tasks are scheduled).

The overhead of the "Tolerating Node Failure" and the "Tolerating Peer Connection Failure" mechanisms is also not significant, because these mechanisms maintain only one small extra data structure on the root node and another one on each non-root node. They do not use additional messages during the normal operation.

4. Reducing the Loss of Resources after *Stop* Failures. In this section, we describe two quite simple extensions of our basic fault tolerance mechanisms by which the system is able to reduce the loss of resources after some kind of *Stop* failures.

4.1. Reconnection after Connection Failure. We have extended the basic fault tolerance mechanisms described in the previous section such that the root tries to recreate the connection to a disconnected node before it starts the migration of tasks in the "Tolerating Node Failures" mechanism.

If a node i observes that the root has become unreachable, it waits for a connection request from the root for a limited time period. If the request does not arrive, i shuts itself down. When a root observes node i becomes unreachable, it tries to reconnect to i in the same way in which a node creates a peer connection to another node during normal execution. If it does not get back any reply, then it starts the "Tolerating Node Failures" mechanism. If i receives a connection request from the root, it sends back a positive acknowledgement (independently whether it has already observed the connection failure or not). When the root gets this request, it informs all other nodes about the connection failure and the successful reconnection to i.

There is a main problem that all nodes (root and all others) now have to deal with: the management of those messages that were sent and might be lost. For solving this, the root can resend some task, result and wait messages to node i, node i can resend some task, result, store and wait messages to the root or some other nodes, and all other nodes can resend some result and wait messages via the root to node i.

For resending the corresponding wait messages, every node can use a set W_j which is used and maintained in Distributed Maple. W_j contains all wait messages

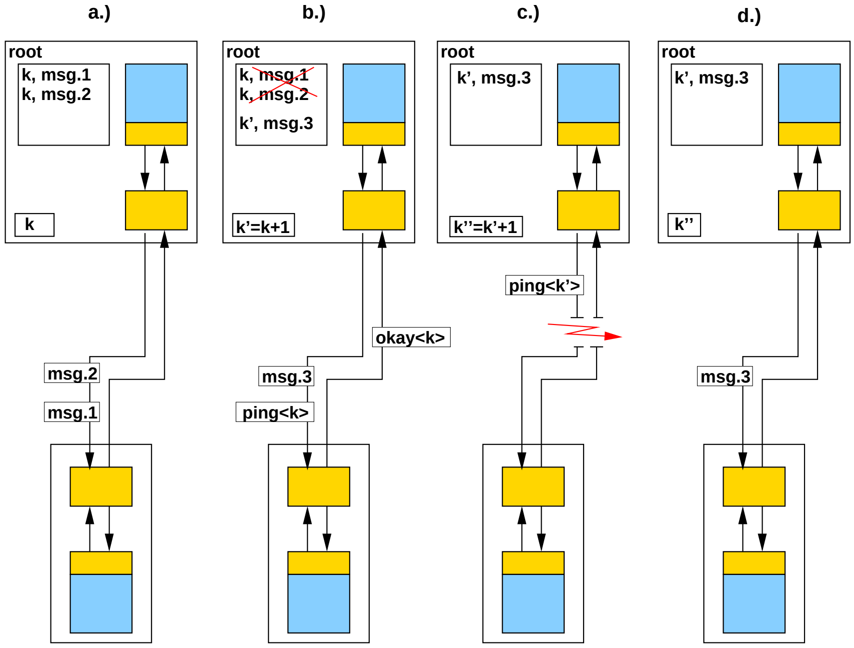

FIG. 4.1. *The resending method:* ***a.)*** *Sent messages are stored in a table.* ***b.)*** *If the acknowledgement message has arrived for the* ping *message, the corresponding messages are removed from the table.* ***c.)*** *The connection fails. The acknowledgement message for the ping does not arrive within a certain time bound.* ***d.)*** *After the reconnection, the messages in the table are resent.*

sent to j; for a subset W_j^r the answers with results have already arrived from j. After j reconnected, each wait message in $W_j - W_j^r$ has to be sent again.

For resending the corresponding task, result, and store messages, every node maintains another set M_j, which contains all task, result and store messages sent to j (see Figure 4.1a). On the root, this set contains also those result messages which were sent by any other node to j through the root. For a subset M_j^a, acknowledgements have already arrived from j (see Figure 4.1b). For acknowledging these messages, the system uses the modified ping message and its acknowledgement. Namely, if the number of non-acknowledged messages reaches a certain bound in M_j, a `ping:`$<k>$ message is sent and the set $M_{j,k}$ becomes $M_j - M_j^a$ where k uniquely identifies this `ping:`$<k>$ message (such a message is also sent in the original case, see Section 3). If an acknowledgement arrives with index k, every element of $M_{j,k}$ is added to M_j^a. After j has been reconnected, each message in $M_j - M_j^a$ has to be sent again (see Figure 4.1c and d).

It may occur that some message arrives twice. This is not a problem, because the system tolerates redundant messages.

4.2. Restarting after Node Failure. We have implemented a quite simple extension of the "Tolerating Node Failures" mechanism by which the root tries to restart the crashed or aborted nodes: after the unsuccessful reconnection to node i, the "Tolerating Node Failures" mechanism is started. The root also starts an asynchronous thread to try to restart eventually i in the same way as in initial phase. If this is managed, node i gets a new identifier instead of i, because all the results that have been stored on i earlier are provided by the root during the rest of the

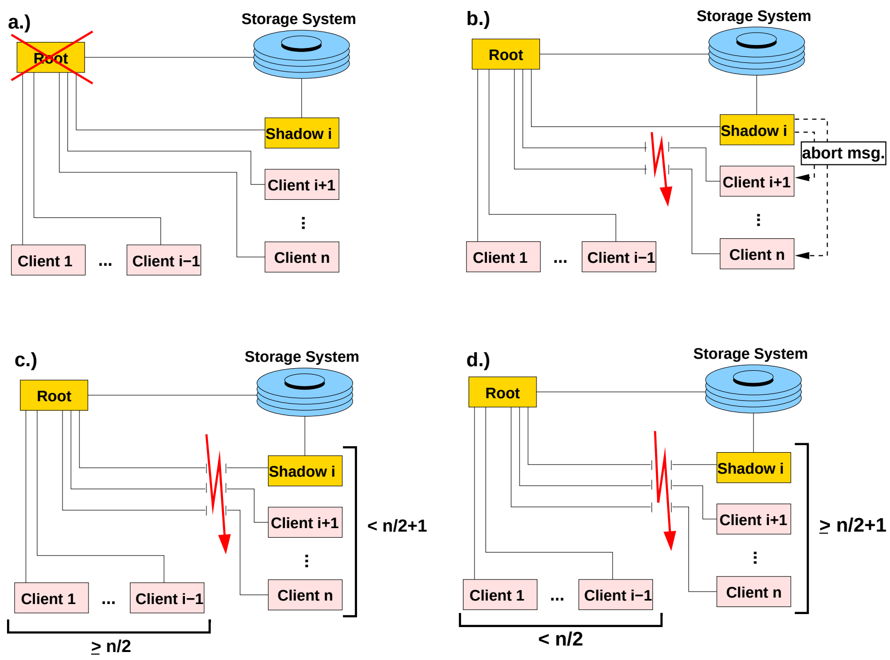

FIG. 5.1. *a.) The root crashes. b.) The connections between the root and some non-shadow nodes break. c.) The network is split. The root is connected to at least n/2 pieces of non-root nodes. d.) The network is split. The shadow node is connected to at least n/2 pieces of non-root nodes.*

execution. The targets of the wait messages have to be determined uniquely from the node identifier contained in the task identifier.

By changing the identifier of the restarted node, we can guarantee that all other nodes interact to the newly started node. Namely, if node i did not fail, just disconnected to the root, it may send some messages to some other nodes. But in this case, these nodes simply drop these messages (because i is declared dead by "Tolerating Node Failures" mechanism). i eventually aborts.

5. Tolerating Root Failure. This mechanism is based on all already mentioned fault tolerance mechanisms (Logging Results, Tolerating Node Failure, Tolerating Peer Connection Failures and Reconnection after Connection Failure) except "Restarting after Node Failure". Therefore, the same assumptions are made as were described for the previous mechanisms: a reliable and stable storage system, reliable communication, no *Byzantine* failures. There are two more important assumptions. First, the storage system is independent from the root and at least one non-root node has access to it. Second, the description of the main task which is performed by the root kernel is initially stored by this storage system.

At no time during the execution of the system, there may exist more than one root. If the root becomes unreachable to another node, this means either the root crashed (see Figure 5.1a) or the connection between the root and the node broke (see Figure 5.1b, c and d). In the second case, the system has to be able to decide about the election of a new root. To guarantee this, the current root always has to be connected to $n/2$ nodes at least (see Figure 5.1c); a non-root node may become the root if and only if at least $n/2$ non-root nodes (beyond itself) accept it as the root (see Figure 5.1d), where n is the initial number of the non-root nodes.

It is possible to use the "Restarting after Node Failure" mechanism together with this mechanism, but the additional restriction is needed: an unreachable node may be restarted if and only if the root has declared it dead (see Section 3) and more than $n/2$ nodes have acknowledged this declaration (the "Tolerating Node Failures" mechanism warrants that every message from a dead node is dropped).

At the initialization of the session, all nodes get the identifier of a special non-root node which can access the storage system. This node is called shadow root or simply shadow. The subsequent explanations assume that there is only one such shadow. In Section 5.5, we will generalize the mechanism to an arbitrary number of (potential) shadows.

5.1. Triggering the Root Election. If the root becomes unreachable to a non-shadow node k and the reconnection time expires, k sends a `root_lost` message directly to the shadow node i. If node i has become also unreachable, k aborts. Shadow i maintains a set R of the identifiers of all nodes that cannot contact the root. When i receives the `root_lost` message from k, it adds k to R. From the moment that the first node is added to R, i waits for a certain period of time. If during this period, no message arrives from the root, i starts the root election. However, if during this time a message from the root arrives (i.e. the root is not unreachable to i), the election is canceled (see Figure 5.1b). In this case, i waits until the root declares k dead (which must eventually happen since k cannot contact the root), and then it sends an `abort` message to k (which causes k to abort, but the root will eventually restart k). Summarizing, root election is only started in the situations illustrated in Figure 5.1a, c and d.

5.2. Performing the Root Election. Shadow node i broadcasts a `check_root` message to all live nodes except those whose identifiers are in R. When a node l receives such a message, it sends back an `acknowledgement` message. Node l also checks the root within a certain time bound. If the root is reachable to l, then l sends back a `root_connected` message to the shadow node i. Otherwise, it sends back a `root_lost` message to i. Node i waits for the `acknowledgement` messages of the `check_root` broadcast for a limited time period and counts them. If this sum plus the size of R is less than $n/2$ where n is the initial number of the non-root nodes, i aborts (see Figure 5.1c). Otherwise, it waits further for `root_lost` and `root_connected` messages and counts them, too. If it receives a `root_lost` message, it adds the identifier of the sender to R (if i observes that a node whose identifier is in R became unreachable, i deletes the corresponding identifier from R). If the number of `root_lost` messages reaches the bound $n/2$, i sends a `new_root` message to all other nodes. If this bound has not been reached, but a `root_lost` or a `root_connected` message has been received from each acknowledged node that is reachable to i, i aborts.

Each node that has received the `new_root` message accepts i as the new root even if the old root is still reachable to it. Summarizing, the shadow node i becomes the new root only in cases represented in Figure 5.1a and d.

In case depicted in Figure 5.1d, the old root eventually realizes that less than $n/2$ nodes are connected and aborts.

5.3. Initialization of the New Root. After the shadow root has become the root, it loads the main task from the storage system and schedules it to its own kernel. Then it declares the old root and those nodes dead which did not acknowledge the receipt of the `check_root` message (when the connected nodes receive this declaration,

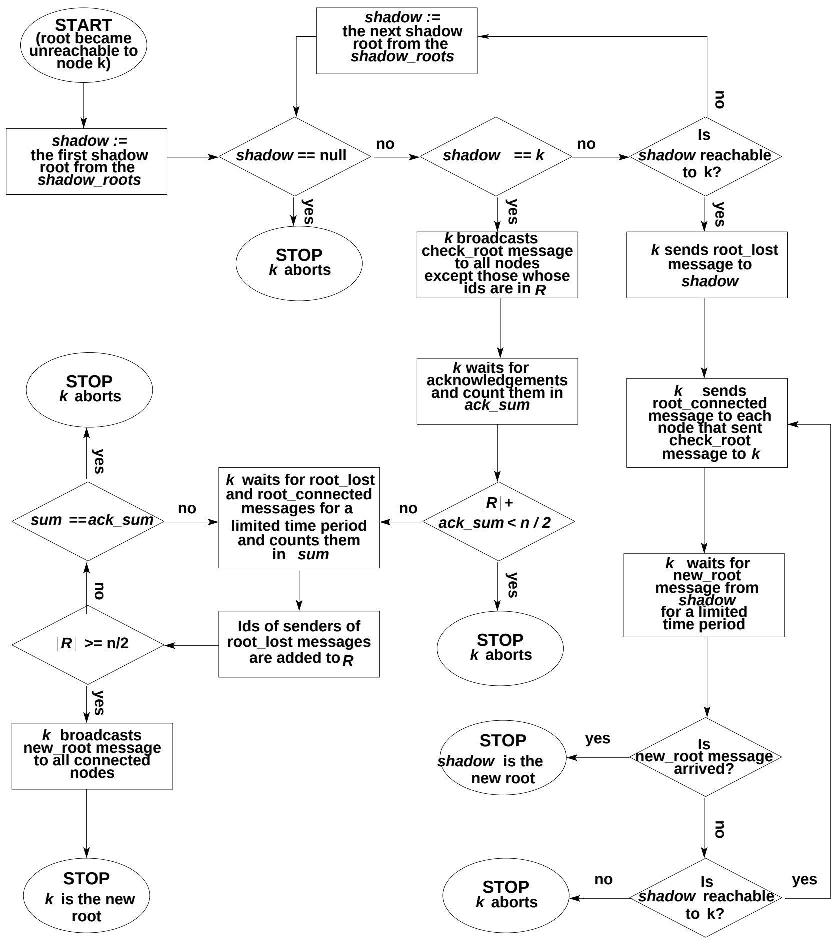

FIG. 5.2. *The generalized root election method on node* k *where* n *is the initial number of the non_root nodes.* R *is a set which consists of the identifiers of those nodes that sent root_lost message to* k. |R| *signs the size of* R.

they resend those wait messages to the new root which might be lost, see Section 3). After a node has accepted a `new_root` message, it resends all store and task messages to the new root which are not acknowledged by the old root (task messages are acknowledged by the root if and only if they have already been logged). It also sends the identifiers of those tasks in a `scheduled_tasks:`<*task_identifiers*> message to the new root which are under processing on this node. The new root keeps these pieces of information in the *scheduled_tasks* table as tuples of a node identifier and a task identifier (this table is the same which are already used by the root in the "Tolerating Node Failures" mechanism in Section 3).

5.4. Task Handling and Termination. If the new root receives a `task:`<*t, d*> message, the logging mechanism tries to restore the result of the task from *d*. If it does not manage, it checks whether *d* may be already logged with a different identifier *t'*. If yes, it checks whether *t'* occurs in the *scheduled_tasks* table. If *t'* occurs in this table, the new root drops this message, because it is already under processing somewhere. Otherwise, it schedules this task to a node.

In the normal case, when there is no root failure during the execution, the termination of the system is always triggered by user interaction. But now that the root of the session is changed, this is not possible any more (because the user interface of the system is located on the initial root, see Figure 2.1a). Therefore, after the shadow root has become the root, it counts separately how many task descriptors and task results are already stored by "Logging Results" mechanism and maintains these pieces of information. If the number of stored task results becomes equal to the number of stored task descriptors, the system terminates (the "Logging Results" mechanism warrants that the descriptor of a task is always stored before its own result and the result of its parent task). In a later session, the system is able to recover the result of the whole computation by the "Logging Results" mechanism.

5.5. Generalization of the Root Election. Above description uses only one shadow node. Now, we generalize the mechanism such that we use a list of the shadow nodes. This list is called *shadow_roots*; all nodes receive it at the initialization of the session.

If the root is unreachable to a node k and the reconnection time expires but the first node in the *shadow_roots* has become also unreachable, k sends a `root_lost` message to the first live node in this list (see Figure 5.2). If such a node does not exist, k aborts. If k has already sent a `root_lost` message to a node and some `check_root` messages arrive from some other nodes, it replies with a `root_connected` messages. If k is the next shadow root, it broadcasts a `check_root` message as described in the previous section.

Finally, the number of the elements of R on each shadow root decides the result of the root election. In the worst case, neither shadow node becomes the root and the whole system aborts.

6. Comparison with a Particular Checkpointing Mechanism. In order to put our work in context, we are now going to compare the fault tolerance features of Distributed Maple with those of systems based on checkpointing mechanisms. As a concrete example, we take the P-GRADE environment into which such a mechanism has been recently integrated [15] (see Figure 6.1). P-GRADE is a parallel programming environment which supports the whole life-cycle of parallel program development [20]. A P-GRADE application is always centralized, where a server coordinates the start-up phase and the execution. The primary goal of the P-GRADE checkpointing mechanism was to make P-GRADE generated applications able to migrate among nodes within a cluster. Its secondary goal was to make periodic checkpoint for supporting fault tolerance.

In the case of the P-GRADE checkpointing, the main idea was to save the states of the processes by making regularly snapshots of them. Thus during the execution a computing phase periodically alternates with a checkpointing phase. As mentioned in Section 1, such an approach is relatively complex because it is targeted to general parallel applications. In the case of our "Logging Results" mechanism, the system saves the computed task results instead of the states of the processes and this saving operation is continuous. This solution is simpler, but it is restricted to parallel programming models based on functional tasks, i.e., tasks that return results without causing any side-effects. Both mechanisms are centralized and transparent to the applications.

P-GRADE makes a checkpoint if and only if an external tool asks the server for the saving of a checkpoint. If the system fails during the computing phase, it can be restarted from the last checkpoint. If the system fails during the checkpointing phase,

	P–GRADE checkpointing	fault tolerance in Distributed Maple
architecture	centralized	centralized
applicability	it is targeted to general parallel applications	it is restricted to parallel programming models based on functional tasks
transparency	transparent	transparent
the saving method	a computing phase is periodically interrupted by a checkpointing phase	continuous
if the system fails during the computing phase	it can be restarted from the last checkpoint	it is able to recognize the corruption of a file; thus it may reuse each properly stored datum in a later session
if the system fails during the check–pointing phase	it can be restarted only from the previous checkpoint	
in case of node failure	it is able to continue execution from the last checkpoint (except if the server fails)	it is able to continue execution by the migration of the tasks (also if the root fails)

FIG. 6.1. *Comparison of Fault Tolerance in Distributed Maple and P-GRADE Checkpointing*

it cannot use any already saved information of this checkpoint and it can be restarted only from a previous checkpoint. The "Logging Results" mechanism saves every task descriptor and task result into separate files and it is able to recognize the corruption of a file; thus it may reuse each properly stored datum in a later session.

If a node fails (except for the server), the P-GRADE is able to recognize this and to continue execution automatically from the last checkpoint. In such a case, Distributed Maple is simply able to continue execution by the migration of the corresponding tasks (also if the root node fails). Distributed Maple regularly checks whether a failed node has rebooted again and in this case restarts the corresponding node process.

Summarizing, the most important advantage of the P-GRADE checkpointing mechanism is that it is made for general parallel applications. But its fault tolerance features are limited in that the main purpose was the dynamic migration of the processes among the node. The main disadvantage of the fault tolerant mechanisms in Distributed Maple that they are special mechanisms and they cannot be used for any parallel computing model. The advantages of our mechanisms are the following: the saving method is continuous; if the system fails in any time, the next session is able to use all properly saved intermediate task results; if any node or connection fails during the execution, the system is able to tolerate it and continue the normal operation without human intervention. Both systems are centralized, therefore the scalability of these systems is an open question.

7. Conclusion. We have implemented in Distributed Maple some fault tolerance mechanisms such that we can guarantee the following: the system does not deadlock and continues normal operation, if any node crashes or some connection between any two nodes breaks; if the system fails after all, we can restart it and continue the computation from a saved state. The improvement of the system is demonstrated by

<table>
<tr><th colspan="2">Stop failures</th><th>without fault tolerance</th><th>using fault tolerance</th></tr>
<tr><td colspan="2">the root node crashes</td><td>system aborts</td><td>system continues the normal operation</td></tr>
<tr><td rowspan="3">a non−root node crashes</td><td>less than half or half of non−root nodes crash within a certain time bound</td><td rowspan="3">deadlock</td><td>system continues the normal operation</td></tr>
<tr><td>more than half of non−root nodes crash within a certain time bound</td><td>system aborts</td></tr>
<tr><td>the root and all shadow root nodes crash within a certain time bound</td><td>system aborts</td></tr>
<tr><td colspan="2">one or more connections between the root and non−root nodes break</td><td>deadlock</td><td>system continues the normal operation</td></tr>
<tr><td colspan="2">one or more connections between non−root nodes break</td><td>deadlock</td><td>system continues the normal operation</td></tr>
<tr><td colspan="2">one or more Maple processes fail</td><td>deadlock</td><td>deadlock</td></tr>
</table>

FIG. 7.1. *Assessment of Distributed Maple Fault Tolerance*

Figure 7.1.

The system can still fail if more than $n/2$ non-root nodes fail, or if the root and all shadow nodes fail within a certain time bound (i.e., if there is not enough time for reconnection).

The system can also fail if neither the root nor one of the shadow nodes has connected at least $n/2$ non-root nodes (this may happen if the network is split to more than two almost equal parts).

There remains only one kind of *Stop* failure situations which may let the system deadlock: if a kernel process fails. To solve this problem, we plan to introduce a new watching mechanism which scans the kernels and restarts them if necessary.

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 (in case of both root node and non-root node) on top does then not require much extra overhead.

In the testing phase of our fault tolerance mechanisms, we executed some long-running computations, whose solutions are required approximately 3 or 4 days. During these execution, we triggered several *Stop* failure situations (both root and non-root failures) off. Our test experiences showed that the system is able to tolerate these kinds of failures and finish the computations.

One reason for this simplicity is the delegation of all logging activities to a single root node that also performs the task scheduling decisions; 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 ($\leq$ 30 nodes); many parallel computer algebra applications do not scale well enough to profit from considerably more nodes.

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] 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.

[3] 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.

[4] 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.

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

[6] 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.

[7] 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 CS Press.

[8] G. E. Fagg and J. 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, Balatonfüred, Hungary, September 10–13, 2000. Springer, Berlin.

[9] 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.*

[10] 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.

[11] Hoon Homg, Andreas Neubacher, and Wolfgang Schreiner, *The Design of the SACLIB/PACLIB Kernels. Journal of Symbolic Computation*, 19:111–132, 1995.

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

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

[14] 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 1991. IEEE CS Press.

[15] József Kovács, Péter Kacsuk, *Server Based Migration of Parallel Applications.* In *Distributed and Parallel Systems – Cluster and Grid Computing, DAPSYS'2002, 4th Austrian Hungarian Workshop on Distributed and Parallel Systems*, pages 30–37, Linz, Austria, September 29 – October 02, 2002. Kluwer Academic Publishers,Boston.

[16] Wolfgang Küchlin, *PARSAC-2: A Parallel SAC-2 based on Threads.* In *AAECC-8: 8th International Conference on Applied Algebra, Algebraic Algorithms, and Error-Correcting Codes*, volume 50 of *Lecture Notes in Computer Science*, pages 341–353, Tokyo, Japan, August 1990. Springer, Berlin.

[17] 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.

[18] M. Migliardi, V. Sunderam, A. Geist, and J. Dongarra, *Dynamic Reconfiguration and Virtual Machine Management in the Harness Metacomputing 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 Mexico, December 8–11, 1998. Springer.

[19] Nancy A. Lynch, *Distributed Algorithms.* Morgan Kaufmann Publishers, Inc. San Francisco,

California, 1996.
[20] *P-GRADE environment*: http://www.lpds.sztaki.hu/projects/pgrade.
[21] Taesoon Park and Heon Y. Yeom, *Application Controlled Checkpointing Coordination for Fault-Tolerant Distributed Computing Systems. Parallel Computing*, 24(4):467–482, March 2000.
[22] 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.
[23] 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.
[24] Fred B. Schneider, *Implementing Fault-Tolerant Services Using State Machine Approach. ACM Computing Surveys*, 22(4):299–319, December 1990.
[25] Wolfgang Schreiner, *A Para-Functional Programming Interface for a Parallel Computer Algebra Package. Journal of Symbolic Computation*, 21(4–6):593–614, 1996.
[26] 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.
[27] 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.
[28] Wolfgang Schreiner, Christian Mittermaier, and Franz Winkler, *Analyzing Algebraic Curves by Cluster Computing.* In *Distributed and Parallel Systems – From Instruction Parallelism to Cluster Computing, DAPSYS'2000, 3rd Austrian Hungarian Workshop on Distributed and Parallel Systems*, pages 175–184, Balatonfüred, Hungary, September 10–13, 2000. Kluwer Academic Publishers, Boston.
[29] Wolfgang Schreiner, Christian Mittermaier, and Franz Winkler, *On Solving a Problem in Algebraic Geometry by Cluster Computing.* In *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.
[30] Wolfgang Schreiner, Károly Bósa, Gábor Kusper, *Fault Tolerance for Cluster Computing on Functional Tasks.* Euro-Par 2001, 7th International Euro-Par Conference, Manchester, UK, August 28 – August 31, 2001. Lecture Notes in Computer Science, Springer, Berlin, 5 pages, Springer-Verlag.
[31] Wolfgang Schreiner, Christian Mittermaier, Károly Bósa, *Distributed Maple: Parallel Computer Algebra in Networked Environments.* Journal of Symbolic Computation, volume 35, number 3, pp. 305-347, Academic Press, 2003.
[32] 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.