JOHANNES KEPLER
UNIVERSITÄT LINZ
Netzwerk für Forschung, Lehre und Praxis

Fault Tolerance for Distributed Maple

Dissertation

zur Erlangung des akademischen Grades

Doktor der Technischen Wissenschaften

Angefertigt am *Institut für Symbolisches Rechnen*

Betreuung:

Erster Begutachter: A. Univ.Prof. Dipl.-Ing. Dr. Wolfgang Schreiner
Zweiter Begutachter: A. Univ.Prof Dr. Josef Küng

Eingereicht von:

Károly Bósa, Dipl.-Ing.

Linz, September 2004

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Dissertation selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Károly Bósa
Hagenberg, 30. September 2004

Abstract

Distributed Maple is a Java-based system for implementing in distributed environments parallel programs in the computer algebra system Maple. It has evolved from Dr. Wolfgang Schreiner's experience in the development of parallel computer algebra environments and from learning from the work of other researchers. As the problems to which the system was applied became more and more complex, the meantime between session failures became a limiting factor of the applicability of the system. However, the fact that the parallel programming model of the system is basically functional gave the chance to develop new fault tolerance mechanisms for Distributed Maple which are more effective than existing solutions targeted to general parallel applications (like checkpointing).

In this thesis, we present and describe how we have extended Distributed Maple with fault tolerance such that the time spent in a long running computation is not any more wasted by the eventual occurrence of a failure. First we introduced a mechanism for the logging of task return values and of shared object values such that after a failure a newly started session can (transparently to the application program) reuse already computed results. Then we concentrate on node failures and permanent connection failures. We implemented some new mechanisms by which a session is able to tolerate connection and node failures (even if the root node fails) without overall failure and continue normal operation. Furthermore, the system periodically attempts to restart the failed nodes and to reestablish the broken connections. Together these fault tolerance mechanisms allow to run computations that take much longer than the meantime between session failures.

With these developments, Distributed Maple is by far the most advanced system for computer algebra concerning reliability in distributed environments.

Zusammenfassung

Distributed Maple ist eine Java-basierte Software, um im Computeralgebra-System Maple parallele Programme für verteilte Umgebungen zu schreiben. Die Software wurde aus Dr. Wolfgang Schreiners Erfahrungen mit der Entwicklung paralleler Computeralgebra-Umgebungen und aus der Arbeit anderer Forscher heraus entwickelt. Als die Probleme, auf die Distributed Maple angewendet wurde, immer komplexer wurden, entwickelte sich die mittlere Zeit zwischen Systemfehlern zu einem einschränkenden Faktor bei der Anwendbarkeit der Software. Allerdings gab die Tatsache, dass das parallele Programmiermodell von Distributed Maple im wesentlichen funktional ist, die Chance, neue Mechanismen zur Fehlertoleranz zu entwickeln, die effektiver als existierende Lösungen sind, die auf allgemeine parallele Anwendungen abzielen.

In dieser Arbeit zeigen wir, wie wir Distributed Maple um Mechanismen zur Fehlertoleranz erweitert haben, sodass die Zeit, die mit langen Berechnungen verbracht wird, nicht letztendlich durch einen Programmabbruch vergeudet wird. Zuerst führen wir einen Mechanismus ein, um die Ergebniswerte von Aufgaben und die Inhalte von Objekten so zu speichern, dass nach einem Programmabbruch ein neuer Programmlauf diese Werte wieder verwenden kann, ohne sie neu zu berechnen (wobei die Anwendung keinen Unterschied merkt). Dann konzentrieren wir uns auf das Ausfälle von Knoten und Verbindungen im verteilten System. Wir haben einige Mechanismen entwickelt, mit denen solche Ausfälle (sogar der Ausfall des Zentralknotens) toleriert werden können, ohne dass das gesamte System abbricht sondern mit der Arbeit normal fortfahren kann. Weiters versucht das System periodisch, ausgefallene Knoten neu zu starten und ausgefallene Verbindungen wieder her zu stellen. Zusammen erlauben es diese Mechanismen, Berechnungen durchzuführen, die viel länger als die mittlere Zeit zwischen Systemfehlern benötigen.

Mit diesen Entwicklungen ist Distributed Maple das weitaus forgeschrittenste System für Computeralgebra, was die Zuverlässigkeit in verteilten Umgebungen betrifft.

Acknowledgements

First of all, I would like to thank my supervisor Wolfgang Schreiner. His patient guidance, inspiration and helpful suggestions have played a significant role in the fact this work has resulted in a Ph.D. thesis. I greatly appreciate the way he combined, on the one hand, giving advices and assistance, and on the other hand, giving much freedom in the research. Moreover I am grateful to him, because he has revised and improved my written English tirelessly in the last couple of years. I hope his effort was not futile.

I want to express my gratitude to my fiancee Anett, who gave me all her love and understanding. Without her support and patience this work would never have been completed.

Finally, I am grateful to my parents for their moral support and encouragement.

The work on the thesis was supported by the Research Institute for Symbolic Computation (RISC), the Upper Austrian Government (Ph.D. scholarship) and by the Austrian Science Foundation (FWF) under the project SFB P1302.

Contents

Chapter 1

Introduction

1.1 Goals and Motivation

During the last decade, the field of distributed computing has considerably evolved. However, as the use of distributed computing systems for long-running and large scale computations is growing, they become increasingly vulnerable to failures. Therefore, an indispensable condition for the extension of the usability of such systems is to increase their reliability. For achieving this, some *fault tolerance* mechanisms have to be adopted such that applications can withstand the number of failures before normal operation is impaired.

In this thesis, we focus on the new fault tolerance ability of Distributed Maple. This software is a Java-based system for implementing in distributed environments parallel programs in the computer algebra system Maple [58]. In contrast to systems for message-based parallel programming, Distributed Maple is fundamentally based on the concept of *functional tasks* that return a result and have no other side-effect. Still this model is very general, e.g., it allows tasks to start other tasks, to refer to the handlers of other tasks as arguments and to return task handlers as results (thus it supports general communication structures). A comprehensive survey on the system and its applications is given in [66].

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. We therefore started to investigate how to extend the time scale of distributed computations to many days as is required for realistic applications.

Our goal for the thesis was the following: by taking advantage of the fact that the parallel programming model of the system is basically functional, we intended to develop new fault tolerance mechanisms for Distributed Maple which are more effective than existing solutions targeted to general parallel applications. Another requirement was that the implemented fault tolerance has to be *transparent* to the application.

To achieve this goal, we invented and introduced some new fault tolerance mechanisms to Distributed Maple [64, 65, 8, 9, 10] by which the system can continue normal operation in the following failure situations: any node (be it a non-server node or be it the server itself) fails or a connection between any two nodes breaks. To cope with the possibility that the systems fails due to other reasons, another mechanism [64, 65, 8] continuously saves the intermediate computing results during the execution; thus we can restart the system and continue the computation from a saved state.

As the outcome of our work, the applicability of the system is extended by the developed fault tolerance mechanisms. This work has considerably profited from the deliberate restriction to the programming and execution model mentioned above. Thus Distributed Maple has become the most advanced system for computer algebra concerning reliability in distributed environments.

1.2 Structure and Originality of the Thesis

In the followings, we present how the rest of the thesis is organized.

Chapter 2 (State of the Art) gives a survey on the fields that are relevant to the thesis. *Section 2.1* gives an overview on the use of Maple for parallel computer algebra. *Section 2.2* describes ongoing work on fault tolerance in distributed systems.

Chapter 3 (Distributed Maple) gives an overview about the original Distributed Maple (without fault tolerance). *Section 3.1* introduces into the usage of the system and collects the commands of the user interface. *Section 3.2* gives a short overview about the system architecture and the execution model of Distributed Maple. These two sections are based on the research work of Dr. Wolfgang Schreiner. At the end of the chapter, *Section 3.3* analyses the possible failure situations which may occur in Distributed Maple.

Chapter 4, Chapter 5, and Chapter 6 describe the achievements of the thesis. The explanations of the fault tolerance mechanisms in these chapters are usually divided to three parts:

- The section “Assumptions and Guarantees” determines what kind of conditions have to be satisfied for the proper working of the mechanism and what kind of guarantees are provided by this mechanism in exchange.
- The section “Algorithm” gives a general description of the fault tolerance mechanism.
- the section “Implementation” presents how the mechanism has been implemented in Distributed Maple and lists the corresponding data structures (implemented as Java classes).

Chapter 4 (The Logging Mechanism) presents the “Logging” mechanism which we introduced to reuse already computed results after a session failure. This mechanism is explained in three stages. In *Section 4.2*, the basic functionalities of the “Logging” mechanism are presented (these are collectively called the *Fast Mode*). This “Fast Mode” was developed by the author jointly with Gábor Kusper and Dr. Schreiner. Later, it has been refined repeatedly by the author. In *Section 4.3*, the “Logging” mechanism is extended to the *Safe Mode* which allows tasks to refer each other in their descriptions and results. This Safe Mode was elaborated only by the author. *Section 4.4* shows how shared objects are handled in the “Logging” mechanism. The handling of the shared object was designed and accomplished by the author together with Gábor Kusper.

Chapter 5 (Tolerating Connection Failures) deals with the tolerance of connection failures in Distributed Maple. In *Section 5.1* the “Reconnecting” and in *Section 5.2* the “Tolerating Peer Connection Failures” mechanisms are described. These two mechanisms were invented and implemented by the author.

Chapter 6 (Tolerating Node Failures) explains three mechanisms by which Distributed Maple sessions are able to tolerate node failures. All three mechanisms were discovered and implemented by the author. *Section 6.1* introduces the “Tolerating Non-Root Node Failures” mechanism first by which tasks can be migrated such that a session may tolerate the failure of individual (non-root) nodes without overall failure. In *Section 6.2*, this mechanism is extended with the “Restarting after Node Failures” mechanism which is able to reduce the loss of resources after node failures. *Section 6.3* describes the “Tolerating Root Failure” mechanism that allows the system to cope with root node failure without overall failure.

Chapter 7 (Examples and Test Experience) collects the new Distributed Maple commands which belong to the fault tolerance features in *Section 7.1* and shows by some examples in *Section 7.2* and in *Section 7.3* how they can be used. Then it analyses the performance of the fault tolerance mechanism in *Section 7.4*.

Chapter 8 (Conclusions) summarizes our results. *Section 8.1* compares the fault tolerance features of Distributed Maple with a checkpointing mechanism recently developed by another group for a particular parallel programming environment. *Section 8.2* outlines further development directions.

Chapter 2

State of the Art

In this section, we sketch a survey of those research areas which are relevant to the thesis.

2.1 Parallel Computer Algebra with Maple

Distributed Maple has evolved from Dr. Schreiner's 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 [39] for computer algebra on a shared memory multiprocessor. The model was refined in PACLIB [31] on which a para-functional language was based [57]. In the following, we give a short overview about some research which is more or less directly relevant for our work. This section is mostly epitomized from [66].

Many papers on parallel computer algebra can be found in [20, 73, 30, 29]; summaries are also available in [54] and [26].

An early approach to use Maple as an engine for parallel computer algebra was "*Sugarbush*" [15] that used the coordination language Linda to manage concurrent activities on multiprocessors and computer networks. The system was used e.g. for parallelizing big number arithmetic [16]; because of its special kernel it fell out of use with decline of Linda support.

Without any special parallel programming support, [71] used Maple in a workstation cluster to parallelize the characteristic set algorithm. Maple kernels were directly started on various machines and communicated by reading and writing to files in a global network file system. Because of the large process granularity, this approach nevertheless achieved good speedups.

The ||MAPLE|| ("*Parallel Maple*") environment [67] used the Guarded

Horn Clause language Strand for coordinating the activities of multiple Maple kernels running on multiprocessors and computer networks. Since the system depended on a special version of the Maple kernel, it could not be distributed; with the decline of Strand, it fell out of use.

[13] describes an environment where Maple computations can be distributed across a network of workstations by use of the DSC system [18] which on the basis of standard Internet services ships source code and input data to computers for execution and retrieves the produced output. The system was used with good success for the parallelization of various polynomial algorithms.

The parallel Maple system described in [6] extended the kernel by message passing primitives for the Intel Paragon distributed memory multiprocessor and provided a corresponding Maple programming interface. This framework only allowed the main program to create other tasks (no nested parallelism). The system was not distributed and is not use any more.

FoxBox [19] provides via a Maple interface access to parallel polynomial factorization algorithms implemented in C++ on top of the message passing library MPI. This is a complementary approach to Distributed Maple and the systems mentioned above, because it allows Maple to use an external parallel program but not to write a parallel program that uses Maple.

The computer algebra system "*muPad*" is on the surface similar to Maple. Its kernel can be dynamically extended by a package for "macro parallelism" implemented in PVM [43]. This package allows to write master-worker programs for distributed environments based on the concepts of message passing, global variables and work groups. The parallel programming model is on a considerably lower level than Distributed Maple.

PVMaple was developed for solving systems of differential equations [48]. Similar to the Intel Paragon Maple and to muPad, it provides a Maple interface for writing message passing programs. However, inspired by Distributed Maple, it uses an external process for performing the actual interprocess communication on top of PVM; Maple and this process communicate via shared files. The system runs on clusters of PCs under MS Windows.

Recently, the authors of PVMaple have produced the prototype of a grid-wrapper for Maple called *Maple2g* [49, 50]. Its essentially core is a proxy software that connects to the Globus middleware such that a Maple master process can use Globus to submit computational jobs to computational nodes. Maple processes may cooperate by message-passing (similar to PVMaple) on the basis of the Java-based MPI implementation mpiJava. Initial tests performed with a small number of grid nodes indicate that reasonable efficiency can be obtained in such scenarios. However, it was also found that local cluster computations based on native MPI tools for communication were substantially faster than grid-based computations using the Globus-based counterparts. Maple2g is restricted to one-level master-slave relationships between nodes; it is planned to generalize these relationships to allow hierarchical grid-based applications.

2.2 Fault Tolerance in Distributed Systems

2.2.1 Failure Detection in Asynchronous Distributed Systems

In practice, one of the major differences between distributed and sequential systems is that in a distributed session, there may be possibilities to avoid overall failure and to continue the entire computation even if some individual components fail. In our days, the field of fault detection and recovery techniques is a very complex and crucial area of distributed computing. Unfortunately, the capabilities of general fault detection mechanisms are very restricted in asynchronous environments by *Fischer-Lynch-Paterson's (FLP) impossibility result* [22].

The FLP impossibility result deals with *Consensus* problems (like leader election, voting or failure detection in distributed system), in which all functioning processes have to propose and agree on a value (called decision value). Roughly, the FLP impossibility result says the following: if we consider a system of completely asynchronous processes (or nodes), where

- we cannot make assumptions about relative speeds of the processes or the speed of communication,
- but we assume that the communication is reliable and at most one node may fail (stop permanently) at any time,

then we can never determine if a node/process has died or if it is just very slow in sending messages. The main corollary of this result is that any kind of Consensus problem has no solution in totally asynchronous systems under these conditions.

Therefore, developing fault tolerance for an asynchronous system requires making some assumptions about the system or about the kinds of faults which can be handled. Usually one assumes an upper bound in communication and processor speed, and declares a process dead if it does not respond within a certain time.

The concept of *unreliable failure detectors* introduced by [14] gives a minimal set of properties that a failure detector must have for solving the Consensus in asynchronous distributed systems. An unreliable failure detector may also make a mistake in detecting a failed process, but it is able to correct this error at a later time. This is guaranteed as long as the unreliable failure detector meets the following two requirements [14]:

Completeness There is a time after which every process that crashes is permanently suspected by some correct process.

Accuracy There is a time after which some correct process is never suspected by any correct process.

This technique has some practiced advantages. For instance, if such a failure detector is distributed (this is the usual case) and each node can access to its own detector, then detectors do not need to agree in the detected failed nodes. Furthermore, the failure detection service does not have to be centralized. Therefore, this concept is ideal for large scalable distributed architectures, like Globus [68].

2.2.2 Fundamental Fault Tolerance Techniques

There exist various fundamental techniques to tolerate detected failures in distributed systems [7, 34]:

- To 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* of resources or services on multiple servers such that, if server fails, a request is handled by another one. Since parallel programming systems are concerned about optimal use of resources, only few systems apply this approach [1, 3] (see also the description of *primary backup systems* in the next section).

- 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 restarted 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 [52, 56]. Some systems perform checkpointing transparently to the application, often on top of PVM [17, 12, 28, 40, 37] or MPI [55]. Other systems rely on application support for checkpointing [5, 46].

- 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 on the application-level. Examples of such systems are the Globus toolkit [68], the Harness environment [44], PVM [27] or FT-MPI [21].

All these approaches are relatively complex because they have to deal with general parallel computations; for special problems much simpler solutions exist [32]. 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 [4], the *functional* programming model provides this potential to a large degree [33].

2.2.3 Server Fault Tolerance in Distributed Systems

One of the popular way to achieve server fault tolerance is to apply the approach of *primary backup systems* [11, 53, 74]. Primary backup systems provide fault tolerance capabilities by replicating the state of primary server on one or more backup servers. In case of a primary server failure, one of the backup server is promoted as the new primary server and the connections between the new server and the clients are re-established.

In the recent developments, these backup mechanisms are often mixed with some kind of *transport-level fault tolerance* technique [2, 70, 42]. Such systems use some kind of extension of a transport-level protocol like TCP for two reasons. First, they can reduce the extra overhead of the logging the primary server state during the normal operation by broadcasting the TCP byte stream from a client to primary and backup servers simultaneously. Secondly, they can hide a server failure from the clients, because the modified TCP protocol keeps the connections between the server and the clients alive while a backup server is activated.

The main disadvantage of primary backup systems is that they require extra hardware components, at least a backup server (which is not exploited at all during the normal operation). Hence, this approach gives a suitable solution only for those applications that are important enough to pay for extra hardware (e.g.: e-commerce applications). Another problem with primary backup system is the limited number of tolerated server failure. Namely, if the (last) backup server also fails, the whole session fails (or aborts) independently from the state of the rest of members in the session.

In ST-TCP [42], the TCP byte stream is "tapped" at an intermediate point between the client and the primary server by some kind of physical network interface; byte sequences sent by either the server or a client are forwarded to at least one backup server. The backup servers keep their state consistent with the state of the primary server by executing the same sequence of requests as primary server does. By this, a backup server is able to substitute for the primary server immediately in case of server failure (*active backup server*).

Unfortunately, this kind of server replication technique can be used if and only if operation of the server is deterministic. Replication of non-deterministic servers in this way has not been solved yet. Nevertheless, this mechanism may be also capable to detect a Byzantine failure of primary server since the backup servers receives the responses of the primary server sent to the clients. If the system contains more than one backup server and their generated responses are different from that one which the primary server issued, the faulty server can be determined by some kind of voting algorithm.

Another approach for the automatic handling of server failures is to select a machine as the new root from the the rest of the distributed session. This can be done by a *leader election* algorithm. The specification of leader election in synchronous systems says (roughly) that a system is always able to reach a state in which all operational nodes agree on the leader. In asynchronous systems (with failures) as stated above, this kind of consensus problems is not solvable [22].

In the *Invitation Algorithm*, Garcia-Molna gives a specification based on the idea of groups for solving leader election in asynchronous system [25]. A group is a set of nodes that agree on a leader. This algorithm is used by the group communication system in Amoeba [35] to reconfigure a group after a node crashes.

In the Invitation Algorithm, each node has a unique priority and each is a member of a common group. A node i monitors the leader of the group by sending a message periodically and waiting for a reply. If it does not arrive within a limited time period, i triggers a leader election. In the first step, i makes new a group with itself as the leader and the only member. Then a leader i sends a message `check` to every other node periodically asking whether there exists any other leader. If at least one node replies that it is a leader, the node i suspends its leader election activity for a time that is inversely proportional to its priority (this may help to prevent initiating of concurrent elections). After this time expired, i resumes its started leader election and invites every other leader by a message `merge` to join a new group with node i as leader.

When a leader receives a message `merge`, it forwards it to all members of its own group. Any node j which receives such an invitation (directly or indirectly) accepts it by sending a message `accept` to node i (the proposed potential leader), which acknowledges it with a message `answer`. If j receives this message `answer` within some timeout period, then it joins the new group with i as a leader. Otherwise, j establishes a singleton group with itself as the leader again and starts to send `check` messages.

Unfortunately, the specification of Invitation Algorithm is too strong in some situations [69], because it does not allow to break a connection between two nodes.

Chapter 3

Distributed Maple

Distributed Maple is a Java-based system for implementing parallel programs in the computer algebra system Maple [58]. The starting point of development on Distributed Maple was in 1998 the goal to parallelize parts of the software library *CASA* (computer algebra software for constructive algebraic geometry). CASA has since 1990 been developed by various researchers at RISC-Linz [45] on the basis of the computer algebra system Maple [72].

The system has been employed successfully to develop the parallel versions for various non-trivial methods and applications in 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 [61, 62, 63].

Distributed Maple 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. In [60], we have analyzed the system performance in these environments in large detail.

Since the system does not require any special kernel extensions it should also be portable to any new commercial version of Maple. Furthermore, an interface of the coordination program to another computer algebra system Mathematica has been developed [47]. Starting from the initial design [59], the system has gradually been refined and extended. The follow general description is based on [58, 66].

3.1 Use of the System

The user interacts with Distributed Maple via a conventional Maple frontend (text or graphical), i.e., she or he operates within the familiar Maple environment for writing and executing parallel programs (see Figure 3.1). Maple commands establish a distributed session in which computational tasks can be processed on any connected machine. The following simple example demonstrates the usage of the environment:

```
gemini!5>maple
    |\^/|     Maple V Release 5.1 (Universitaet Linz)
._|\|   |/|_. Copyright (c) 1981-1998 by Waterloo Maple Inc. All rights
 \  MAPLE  /  reserved. Maple and Maple V are registered trademarks of
 <____ ____>  Waterloo Maple Inc.
      |       Type ? for help.
> read`dist.maple`;
Distributed Maple V1.1.15 (c) 1998-2001 Wolfgang Schreiner (RISC-Linz)
See http://www.risc.uni-linz.ac.at/software/distmaple
> dist[initialize]([[condor,linux],[pinwheel,octane]]);
connecting condor...
connecting pinwheel...
                                    okay

> t := dist[start](int, x^n, x);
                                    t := 0
> r := 1 + dist[wait](t);
                                        (n + 1)
                                       x
                              r := 1 + --------
                                        n + 1

> dist[terminate];
                                    okay
> quit;
```

To use the system, the user first has to load the file `dist.maple` which implements the interface to the distributed backend by a Maple package `dist`. By `dist[initialize]`, she or he can trigger the establishing of the distributed backend. In our example, two additional Maple kernels are created on machine `condor` of type `linux` and on machine `pinwheel` of type `octane`, respectively. The machine types are used to lookup the system-specific startup and performance information which is located in a file `dist.systems`.

If the remote processes are successfully started, the user can start the processing of her computational tasks right away by issuing the command `dist[start]`. The command `dist[wait]` blocks the current execution until the corresponding task whose identifier is given in the command as an argument has terminated and then return its result. Finally, the user has to use the `dist[terminate]`. This command closes the distributed session by destroying all remote Maple kernels and closing all network connections.

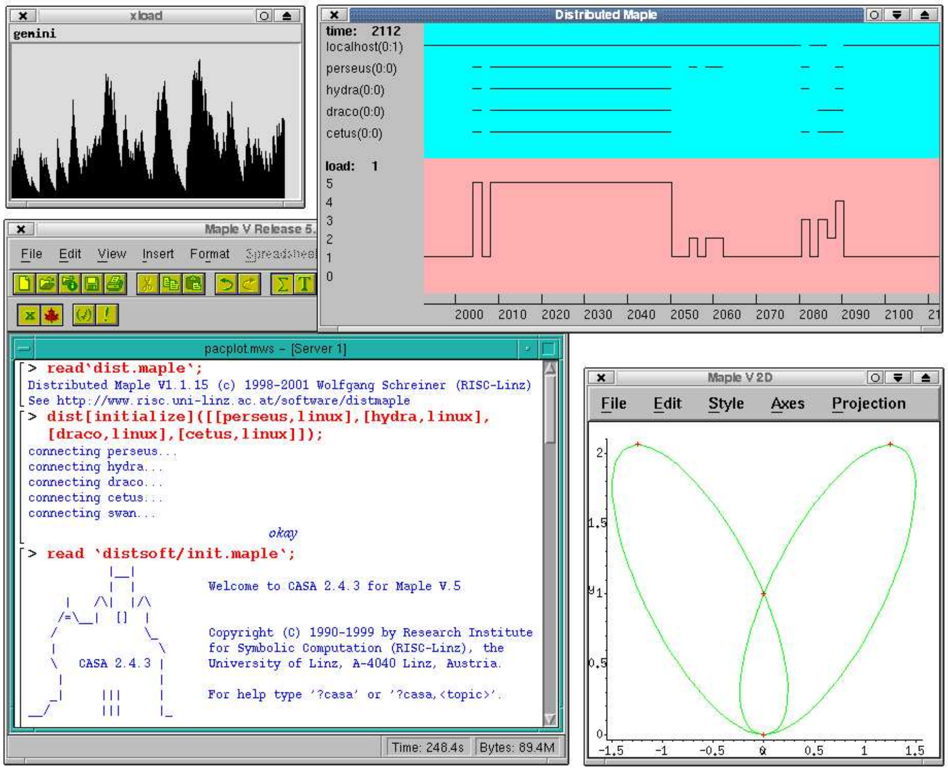

Figure 3.1: *The User Interface of Distributed Maple*

3.1.1 User Manual

In this section, we give a detailed but not complete description of the programming interface in Distributed Maple [58].

Initialization and Termination of a Distributed Session

- `dist[initialize]`(*machineList*) initializes a distributed Maple session by starting remote processes and establishing the corresponding network connections. Each element of *machineList* is a list [machine, type] where machine is the domain name (or IP address) of a machine and *type* is a predefined key which denotes a machine type. By this *type* parameter, the system is able to determine the system specific startup and performance information for every supported platform.

- `dist[terminate]()` terminates the current distributed Maple session by destroying all remote Maple kernels and closing all network connections. This procedure should be called before quitting the interactive Maple session.

- `dist[all]`(*command*) executes *command* on every Maple kernel connected to the distributed session.

Functional Parallelism

- `dist[start]`(*f, a, ...*) creates a task that computes *f(a, ...)* on some Maple kernel connected to the distributed session.

- `dist[process]`(*f, a, ...*) This call creates a task that computes *f(a,...)* on a separate (possibly newly created) Maple kernel. This kernel has executed all the commands issued through previous calls of `dist[all]` in the corresponding order. During the execution of the task, no other task is scheduled on the kernel.

- `dist[wait]`(t) suspends the current execution until task t has completed and then returns its result. While the current execution is suspended, other tasks can be executed by the corresponding Maple kernel.

- `dist[select]`(*taskList*) suspends the current execution until the execution of some task in *taskList* has completed; it then returns a selector for this task, i.e., a list [*index, result*] where *index* denotes the position of the task in the list and *result* is its result.

Shared Objects

- d `:= dist[data]`() creates a shared object d that may be used to communicate data between different tasks. The object is initially empty.

- `dist[get]`(d) returns the value of shared object d; if d is still empty, the current task is blocked until a value is put into d.

- `dist[put]`(d, v) puts the value v into the shared object d and releases all tasks blocked on an attempt to get a value from this object.

- `dist[clear]`(d) resets the shared object d into the empty state. Further attempts to get a value from d blocks the corresponding task.

3.1.2 Visualization

Distributed Maple supports on-line visualization to illustrate the dynamic behavior of a session during its execution. When the user issues a command `dist[visualize]` during the session, a window pops up in which the workload status of each machine and the utilization of the total system resource are displayed (see the window on the right side on the top in Figure 3.1).

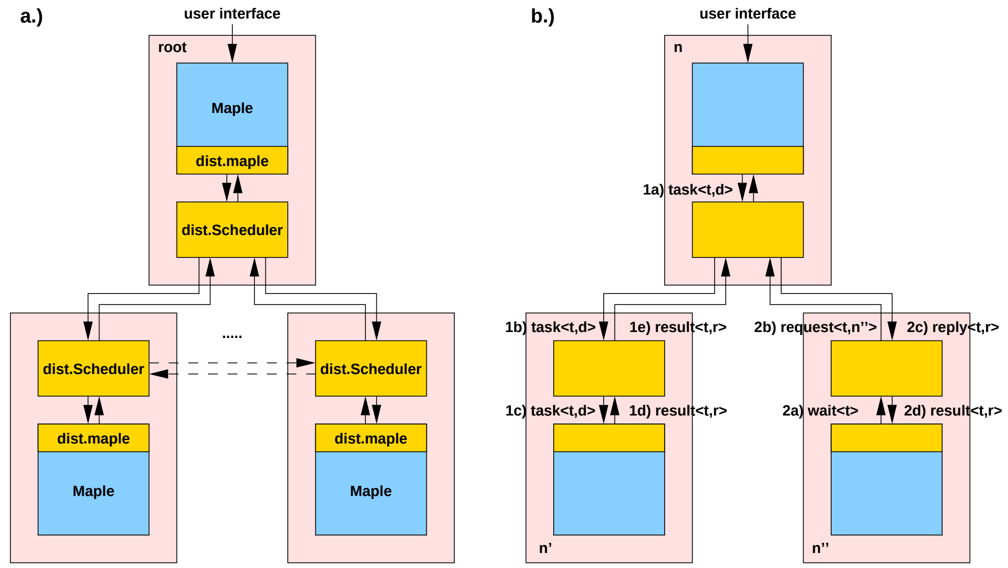

Figure 3.2: ***a.)*** *System Model* ***b.)*** *Execution Model*

This real-time visualization window is split to two parts. Its upper half part shows which machine processes one or more tasks and which one has no any job at the same time (workload). The lower half part of the window displays how many machine works together at the same time (resource utilization).

3.2 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 coordinates node interaction (see Figure 3.2a). 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 3.2b): from the perspective of a scheduler, a kernel on node n 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:`$<t, r>$ to n where r represents the computed result. r is stored in a table *resultTable* on n. 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. 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 an unique identifier t=$<n,i>$ 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 (and on which the task result is stored); this 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''.

By a message `all:`$<$*command*$>$, all nodes can be initialized together, because its argument *command* is executed on every Maple kernel in a session.

3.2.1 The Scheduler

As mentioned before, the scheduler is a Java program which distributes and coordinates the tasks created by all kernels. After establishing of a distributed session was triggered by the user, an initial scheduler process alias root scheduler is created first by the local Maple kernel. Then this scheduler reads all system-specific startup and performance information from a file *dist.systems* and starts the other instances of the scheduler on remote machines. Then each of these remote instances starts and initializes its own local computation kernel.

After a distributed session has been created, every scheduler instance accepts tasks from its local Maple kernel and forwards them to the root scheduler. The root schedules these tasks among all nodes connected to the session by a simple load balancing algorithm (see in Section 3.2.4).

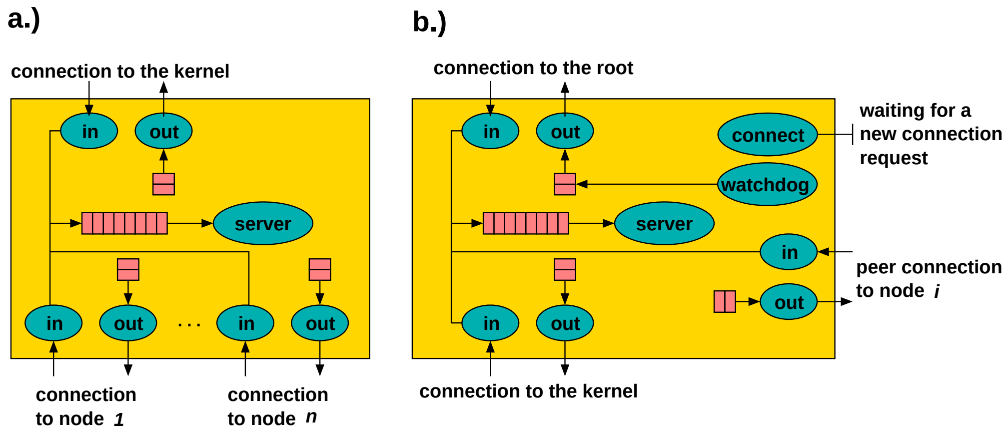

Figure 3.3: ***a.)*** *Scheduler Architecture on the Root* ***b.)*** *Scheduler Architecture on Non-Root Nodes*

The root scheduler consists of a number of concurrent threads as shown in Figure 3.3a. A central *server thread* implements the main functionalities of the scheduler. This thread takes out the received messages from the central input buffer one by one and processes them sequentially. To each input channel, an other kind of thread called *input thread* is assigned that listens on the channel. If a message arrives from a channel, the corresponding input thread puts it into the central input buffer.

To send a message, the sender (thread) has to put the message into the corresponding output buffer. Namely, each output channel has an own buffer (there is no central output buffer) and an own thread, too. These threads are called *output thread* and their task is to read the message from the output buffer and to write it on the output channel.

The architecture of non-root schedulers is almost the same, but they have two additional threads called *connect thread* and *watchdog thread* (see Figure 3.3b). The role of connect thread is to wait for peer connection requests from other non-root nodes and to establish input/output channels for a peer connection after such a request (see for more details in Section 3.2.2). In the watchdog thread, the mechanism *watchdog* is implemented (see Section 3.2.3).

3.2.2 Peer Connections

Initially, there exists a connection only between the root (the local node) and each remote node. However, all nodes know of each other, i.e., of a machine address and the number of a port on which a connect thread is listening for connection requests. When a node needs to send a message to one of its peers, thus a direct connection is established for the message transfer. These connections remain persistent through the rest of the session.

3.2.3 The Watchdog Mechanism

The only mechanism originally available in Distributed Maple for handling faults is a *watchdog* mechanism on every non-root node that regularly checks whether a message had been recently received from the root. If not, it sends a `ping` message to the root that has to be acknowledged. If no acknowledgement arrives from the root within a certain time bound, the node shuts itself down.

3.2.4 Load Balancing

A core problem of distributed system is to exploit the system resources most efficiently by load balancing. The various load balancing techniques tries to reduce the global execution time by assigning statically or dynamically more workload to more powerful resources. Static load balancing attempts to determine appropriate shares of work to be distributed to processors at application startup, while dynamic load balancing may redistribute some part of the work during the execution in order to compensate for differences in processors performance [38].

Distributed Maple has a very simple static load balancing algorithm which is able to measure neither the performance nor workload of the processors. Namely, to initialize a distributed session, the user has to give names or IP addresses of all machines of which the session consists and a predefined key for each machine which denotes a machine type (e.g.: linux, octane, origin, ultrasparc, etc.). By these machine types, the root scheduler can find (among others) some system-specific performance information in a file `dist.systems` for each machine. These are the followings:

- the relative *speed* is a performance factor of a machine type,
- the *maxload* denotes the maximum number of executable but not yet started tasks that a kind of machine may hold at a time,
- and the *minload* denotes the minimum number of executable but not yet started tasks that a kind of machine should hold at a time.

The root scheduler always distributes the task among the nodes by using these static performance factors independently the real workload state of processors at a time. Of course, this solution is less efficient than a load balancing according to the real workloads (e.g.: some other users or applications maybe use the corresponding machines, too), but it is still sufficient for our current requirements.

Roughly, the root scheduler first attempts to schedule a task to the fastest node on which there is no any task under processing. If there is no such a node, the root scheduler tries to find the fastest node on which the number of scheduled but unfinished tasks is less than the *maxload*. If the root scheduler does not find such a node, it schedules the task to the queue of own local kernel. If the number of scheduled but unfinished tasks on a node becomes less than the *minload*, the node notifies the root scheduler about this fact.

3.2.5 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 a scheduler on some node n, the local kernel may issue a message `data:`$<t>$ 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 not else as an result descriptor that is stored on node n (on the node where it was created).

If we compare the shared objects with tasks, the only difference is that a shared objects receives its value r by a message `put:`$<t, r>$ instead of `result:`$<t, r>$. Any task that knows t may issue a put message, *if the corresponding shared object is still empty.* However if a kernel on some node n' needs the value r, it also sends a message `wait:`$<t>$ to node n as in the case of requesting task results.

3.3 Failure Analysis

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 system faults may happen in any case.

All of the possibly fault cases mentioned above can be classified into the following failure types [41]:

Communication failures This kind of failures cover those situations in which a message is lost or duplicated or becomes corrupted.

Stop failures A session or a part of a session (e.g.: a kernel process, a scheduler process, or a node) can stop somewhen in the middle of its execution by several reasons.

A connection may also interrupted such that communication becomes impossible at all (permanently or for a limited period).

Byzantine failures A faulty process (or machine) can exhibit completely arbitrary behavior.

We do not deal with the tolerating of Communication failures and Byzantine failures (our system currently is not able to handle corrupt, lost or duplicated messages and corrupt task executions). In the functional parallel programming model, the Communication failures may be handled similarly as in the message passing or any other general parallel programming model; there exist already numerous general and effective algorithms for tolerating of this kind of failures [41].

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 aborts,
- 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.

Chapter 4

The Logging Mechanism

The first step towards fault tolerance is to store from time to time consistent states of a computational session. If the session fails for any reason, the computation does not need to be restarted from the scratch, because it can continue from such a stored intermediate state of the session.

To achieve this goal, one of the most popular general algorithm is *check-pointing*, which regularly saves global snapshots of the session state [17, 12, 28, 40, 37, 55]. The checkpointing algorithm can deal with all side-effects of processes (because it saves the whole state of each node and the sent but not yet received messages). However, the execution phase of the system is periodically interrupted by a checkpointing phase which has some negative consequences for the performance.

The functional programming model of Distributed Maple simplifies the problem, because the tasks are computed and return results without causing any side-effects. Consequently, it is enough to log the results of tasks and values of the shared objects for storing a consistent state of a session. If the session fails for any reason, we can thus start it and read the logged data (transparently to the application program) without re-executing the corresponding tasks.

This "Logging" mechanism [64, 65, 8] is based on the centralized architecture of the system and only requires that the root is connected to a stable storage system. Every task created by any node is forwarded to the root which schedules it for execution on another node (see Section 3.2). After a session failure, the root attempts to recover the corresponding task result before scheduling the task. Thus the recovery process does not need to send additional messages (no extra communication) and a computed task is recovered if and only if its result is needed for the further computation.

However, logging task results needs extra work to be done, because in normal operation, tasks are processed on various nodes and the results are not sent to the root. Hence, we need an extra message to send each result to the root in order to log it.

Furthermore, our "Logging" mechanism is complicated by the fact that task arguments and results may embed task identifiers and that the scheduling layer of Distributed Maple has only a limited amount of information about the activities of the computing layer.

The system supports two kinds of logging modes which are based on different assumptions (see Section 4.1) and can be selected on the startup of a session: Section 4.2 describes the *Fast Mode*, which is quite simple and very effective. Section 4.3 discusses the more general *Safe Mode* which contains some additional operations. Section 4.4 explains how the "Logging" mechanism is extended in order to incorporate shared objects.

4.1 Assumptions and Guarantees

For the correct working of the "Logging" mechanism, our assumptions are the following:

- The root has access to a writable file system on which it can implement a stable storage.
- The communication is reliable between the system components. Messages are never corrupted or duplicated and they may be lost only in *Stop* failure situations.
- All system components work correctly (no *Byzantine* failures).

Please note our assumptions do not exclude *Stop* Failures which may occur in arbitrary periods. If the system satisfies these conditions, we can provide two kind of logging and recovery mechanisms, which are called:

Fast Mode is the base of the "Logging" mechanism. In this mode, we restrict our consideration to programs whose tasks are *first order*: neither any task description nor any task result contains a task identifier, i.e., task identifiers are not passed as parts of task arguments and are not returned as parts of task results.

Safe Mode is the advanced "Logging" mechanism, in which we allow to use *higher-order tasks* (tasks which receive task identifiers as arguments or return them as results).

In both modes, we guarantee, that if the system fails in Safe Mode, we can restart it and we can continue the computation from a saved state. However, human activity is needed for the restarting of the system.

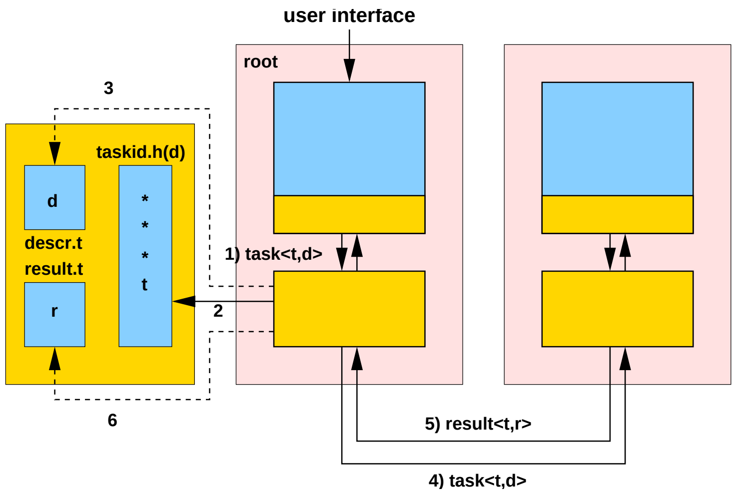

Figure 4.1: *Logging task description and task result in Fast Mode*

4.2 Fast Mode

In Fast Mode, all basic functionalities of the “Logging” mechanism are already realized but methods are missing for handling task identifiers embedded in task descriptions and in task results.

4.2.1 Algorithm

The “Logging” mechanism utilizes the fact that in Distributed Maple the root is in charge of task distribution: every new `task:`$<t,d>$ 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 4.1):

Logging When the root receives a `task:`$<t,d>$, 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:`$<t,r>$ 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:`$<t,d>$ 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:`$<t,r>$ 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.

4.2.2 Implementation

The most important issue which we had to determine at the beginning of the implementation was how to identify logged results. We cannot use task identifiers for this purpose, because they are not the same in different sessions (however, we should store the task identifiers, too, because they are needed for the recovery of *higher-order tasks*, as we can see later in Section 4.3). The only thing which uniquely identifies a task result is its description. Thus, the "Logging" mechanism has to store the task descriptions, too.

Roughly, the mechanism works in the following way. As mentioned before, the root receives all task descriptions created by all nodes. Before the root schedules a task for execution, it tries to restore the result of the task from the storage system (because the result may be computed and stored in a previous session). If it cannot restore it, it saves the description and schedules the task. When the result is computed, the root receives it and saves it to the storage system.

The system stores each piece of data (descriptions and results) in different files, so that the loss of data can be minimized after a I/O failure. The logged data are identified by both the task identifiers and hash codes generated from the task descriptions.

The implementation of this mechanism has to take care of some other issues:

- Two or more task descriptions may have the same hash code.
- The description of a task may not be stored (yet) or it is stored but corrupted.
- The result of a task may not be stored (yet) or it is stored but corrupted.

New Data Structures and Messages

Two additional kinds of messages had to be implemented:

- By a message **stable:<*mode*>**, the logging mechanism can be switched on or off and the active logging mode can be changed. It has only one argument, which is the identifier of such a mode (value **1** switches on the Fast Mode, value **2** switches on the Safe Mode and value **0** switches off logging).
- In a message **store:<*t, r*>**, a computed task result is sent to the root in order to save the result to the storage system. The message has two arguments, the identifier t of a task and the result r of this task.

New Files

The "Logging" mechanism saves all necessary data into a `logging` directory using the following files:

- A file **result.*t*** contains a task result identified by file extension t.
- A file **descr.*t*** contains a task description identified by file extension t. In Fast Mode, this file is only needed to identify a result of a particular task for recovery. But in Safe Mode described in Section 4.3, the system may restore the description of a particular task from this file by identifier t, too.
- The file **taskid.*h(d)*** contains basically the identifier of the task assigned to task description d. The file extension *h(d)* is a numeric hash code computed from d. This file allows of access the corresponding files `result.`*t* and `descr.`*t* by a task description. Unfortunately, we cannot guarantee that a composed hash code is unique for each task description. Therefore, it may occur, that a `taskid.`*h(d)* file contains more than one task identifier (because two or more task descriptions have the same hash code).

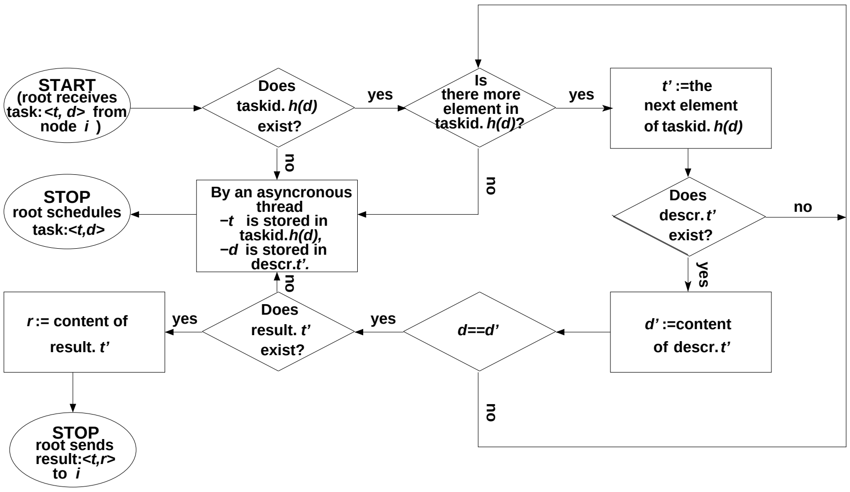

Figure 4.2: *Recovery in Fast Mode*

The system has to be able to check whether a particular file contains valid or corrupted data, because a previous session could crash during the writing of this file. Therefore, every file contains a new line character at its end and the restore mechanism always checks whether the end of a file is a new line character (data cannot contain this character). If this is not the case, the system does not read and restore the content of the file.

Every file is written without any buffering procedure in order to minimize the loss of data in case of a session failure.

Starting Logging in Fast Mode

The user can switch on the Fast Mode by the call `dist[logging](1)`. It is allowed to issue this call only after a session is established (see the description of call `dist[initialize]` in Section 3.1.1).

The effect of this command is that the Maple frontend sends a message `stable:<1>` to the root scheduler. When the root scheduler receives this message, it creates the `logging` directory (if it does not exist) and it broadcasts message `stable:<1>` further to every node. By this notification, every node changes to Fast Mode.

Result Recovery by Task Description

When the root scheduler wants to check whether a result has already been stored for a task description d, it generates a hash code *h(d)* from d and looks for a file `taskid.`*h(d)* (see Figure 4.2). If such a file exists, it takes

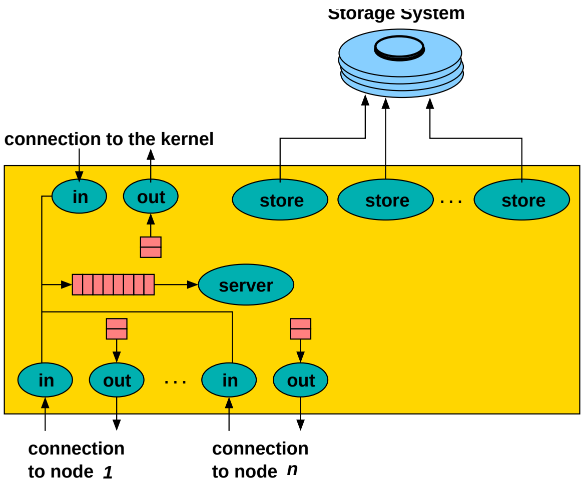

Figure 4.3: *Scheduler Architecture on the Root in Fast Mode*

the first task identifier t contained by this file and it compares d with the content of file `descr.`t. If they are identical, the root scheduler recovers the result contained by `result.`t .

If `descr.`t does not exist, or it exists, but its content is either corrupted or not equal to d, then the root scheduler takes the next task identifier from `taskid.`$h(d)$ and it tries to recover the task result for d by this identifier, too. If there is no more element in `taskid.`$h(d)$, the recovery operation cannot restore the result for description d. The recovery mechanism may be also unsuccessful, if `taskid.`$h(d)$ or a particular result file is missing, or their contents are corrupted.

Modified Task Message Handling

When a scheduler on a non-root node receives a message `task:`$<t,\ d>$ from a local kernel, it forwards this message to the root. If the Fast Mode is active and the root scheduler receives a message `task:`$<t,\ d>$, it checks whether it can restore the result by description d from the storage system (as described in the previous paragraph).

If the recovery is unsuccessful, the root scheduler creates a concurrent thread to save d into a file `descr.`t (separately from the main scheduler functionality implemented in the central server thread, see Figure 4.3). Then the task t is scheduled to a node.

If the recovery is successful, the root restores the result r of the task t and sends it in a `result:`$<t,\ r>$ message to the node which created this task (the identifier of this node can be determined from the task identifier t).

Store Message Handling

When a kernel has computed the result r of a task t, it sends a message `result:`$<t,\ r>$ to the local scheduler. This scheduler forwards this message to the node which created task t and also sends a duplicate of r in a message `store:`$<t,\ r>$ message to the root. If the root scheduler receives a message `store:`$<t,\ r>$, it creates a subsequent concurrent thread to save r into a file `result.`t (separately from the main scheduler functionality implemented in the central server thread, see Figure 4.3).

4.3 Safe Mode

The mechanism described in Section 4.2 guarantees correct results if task arguments and result 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:`$<t'>$ 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 have to guarantee that each task description is always logged to the storage system before other task descriptions or task results which may contain its identifier are logged. The idea is that the description of any task created by a previous session can always be recovered and rescheduled, if another task refers to it by its identifier, but its result is not available. How can we achieve this?

We analyzed the possible relations between the tasks and we managed to determine those situations in which a task t may know the identifier of another task *t'*. These are the following:

- If t establishes *t'* (t is the parent of *t'*).
- If t and *t'* have a common parent and *t'* is established earlier than t. In this case, the description of t may contain the identifier of *t'*.
- If the parent task of *t'* embeds the identifier of *t'* into its result. t may receive this result as the answer for its issued `wait` message.
- At last, if *t'* is the parent of t. *t'* may build in its own identifier into the description of its spawn task t.

We subsequently refined the "Logging" mechanism to satisfy the following conditions:

Condition A A task description may be written to the storage system if and only if all task descriptions previously created by the same node have been completely stored (because the current task description might contain the identifier of such tasks). If the description of the current task were stored earlier than the descriptions of some previously created ones and the system crashed before the descriptions or the results of the previously created tasks were completely written to the storage system, then the next session would refer to some no more existing tasks.

Condition B A task description may be scheduled to a node if and only if the task description has been completely stored to the storage system. Otherwise, this task might create some other tasks whose descriptions contain its task identifier. If these task descriptions were stored earlier than the description of their parent task and the system crashed before the description or the result of this parent task were completely written to the storage system, then the next session would refer to a no more existing task.

Condition C The result of a computed task may be sent to another node if and only if all the task descriptions created by this task have been completely stored to the storage system. Otherwise, the result of the computed task might contain the identifier of its spawn tasks. If another task received this result and created some new tasks description by using of the received identifier (or the result is simple logged) before the description of these spawn tasks were completely written to the storage system, then the next session would refer to no more existing tasks.

4.3.1 Algorithm

In order to distinguish task identifiers of different sessions, we introduce a *session identifier* s. 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 $<s,n,i>$ that embeds the number s of the session in which the corresponding task was created. The mechanism described in Section 4.2 is now generalized as follows:

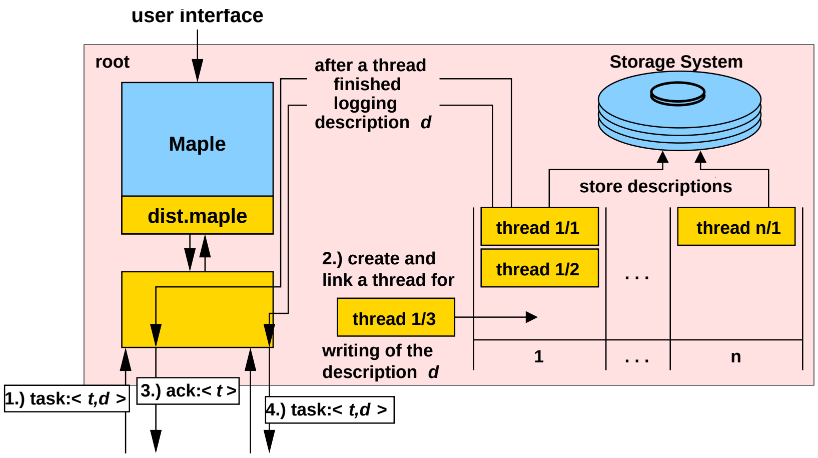

Figure 4.4: *Logging Task Description in Safe Mode*

Logging To support higher-order tasks, the basic "Logging" mechanism described in Section 4.2 has to be extended with some additional functionalities such that the previously mentioned three conditions are satisfied:

- For achieving Condition A, the root maintains a queue for each node (see Figure 4.4). When it receives a `task:`$<t,d>$ and the result for description d cannot be recovered, the root creates a thread to write d to the storage system as in Fast Mode. But instead of starting this thread immediately, the root places it as the last element into the queue of that node from which the message arrived. If this newly inserted thread is the first element of the queue, it is started immediately. Every thread which is the first element of a queue writes a description asynchronously to the storage system. Before a thread finishes its activity, it starts the next thread in the queue. So, the descriptions created on the same node are written sequentially to the storage system. Since the order of messages sent from a node to another does not change during a transmission, we can guarantee that these task descriptions are stored in the order of their creation.
- For achieving Condition B, the root schedules a task to an idle node if and only if logging of the task description is completed. This practically means the previously mentioned threads have to schedule the tasks after they logged the task descriptions to the storage system.
- For achieving Condition C, a result message received from a local kernel is not anymore forwarded immediately to the root or to

the node on which the corresponding task was created (because the description of each spawn task created by the computed task has to be logged first to the storage system). To inform the corresponding node (on which the result message is created) about the completed logging process of a task description, the root uses a new message type called `ack:`$<t>$, where t is task identifier belongs to the logged task description (see Figure 4.4).

Furthermore, for each scheduled task t, the node computing the result of t maintains a set S_t. S_t denotes all tasks created by t; for a subset S_t^a the acknowledgements about completed logging of descriptions have already arrived. If a `result:`$<t,\ r>$ message arrived from the local kernel, where r is the computed result of t, but $S_t - S_t^a$ is not an empty set, then the message will be stored locally. When an `ack:`$<t'>$ is received, where t' is an element of S_t (t' is a spawn task of t), $S_t - S_t^a$ will be evaluated again. The locally stored result of t may be forwarded to other nodes if and only if $S_t - S_t^a$ is an empty set.

Recovery In addition to the actions described in Section 4.2, 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:`$<t>$ where the session identifier of t is not that of the current session, the scheduler on n sends a `request:`$<t,n>$ 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:`$<t,r>$.

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:`$<t,d>$ which is handled as usual. When a kernel issues a `result:`$<t,r>$ of a task created in a previous session, the scheduler forwards this result to the root.

There is one more case which has to be covered to allow higher-order tasks. Assume that the description and the result of a task t is logged into `descr.`t and `result.`t. After a session has failed, the description of a task t' may be identical with the logged description of t in the next session and the result of t is recovered as the result of t'. This means neither `descr.`t' nor `result.`t' files are established. So, if session fails and restarted again, there may exist a task t'' in the newest session whose description or result contains task identifier t' as an argument. In this case, one of the kernels may send a `wait:`$<t'>$ to the root (because t' was created in a previous session), but t' does not exist any more.

The problem is either the corresponding result or at least the corresponding description is available in the storage system, but these informations cannot be assign to t'. Therefore, the root establishes a file `link.`t' when the result of t is recovered as the result of t' by their identical descriptions. This file contains the task identifier of t. So, if a `wait:`$<t'>$ is issued in a later session and root cannot recover either description of or result of t', then it attempts to recover one of them by the content of `link.`t'.

4.3.2 Implementation

In Safe Mode, a task description or a task result may contain some identifier of another task. It may thus occur that a message `wait:`$<t>$ refers to a task identifier t which was created in a previous session. Therefore, such a message is forwarded to the root, which tries to recover the corresponding result. For this, the logged data are identified in the storage system by task identifiers (as described in Section 4.2.2).

During the implementation of this mechanism, we took care of the following cases:

- The corresponding result may already be in the memory of the root (it is more efficient to use the values in the memory instead of reading them from file).
- The result is available in the storage system and we are able to restore by its task identifier.
- The correct result is not available in the storage system, hence, the root tries to recover the description of the task by the task identifier in order to reschedule it and recompute the result.
- Neither the result nor the description of a task are available. In Safe Mode, this situation can occur if and only if a task result is recovered by

a hash code generated from a task description (in this case, the current task identifier is different from logged one), therefore the current task identifier is not stored in the storage system. In a latter session, one of the nodes may refer to this not logged task identifier and the root tries to recover the result by this identifier.

New and Modified Data Structures

- Every task is uniquely identified by its **task identifier** which consists of 3 parts:
  1. A *session identifier* (see below).
  2. A *node identifier* which says which node created the task.
  3. A *task index* which enumerates all tasks created by the node.
- The current session is uniquely identified by a **session identifier** which is part of the identifier of every task and of every shared object. Accordingly, the system always can determine which session created a particular task or shared object (see for more details in the description of the `sessionid` file in this section).
- **clientTable** is a hash table whose keys are node identifiers and whose elements are references to threads. This data structure is located only on the root and it is needed for the synchronized logging of task descriptions (see Figure 4.5).
- **running_tasks** is a list which contains the identifiers of all the tasks which are under processing on the primary Maple kernel of a node at the same time (when a running task executes a command `dist[wait]`, it is blocked and another task can start execution). The head of this list refers to the currently active task. This list is needed when the system wants to determine the parent of a task (see Figure 4.6).
- **externUsed** is a hash table whose keys are the identifiers of the external Maple kernels which are located on the same node and whose elements are the identifiers of the tasks which run on these external kernels. This hash table is needed when the system wants to determine the parent task of a task (see Figure 4.6). An external Maple kernel can be started by a call `dist[process]` (see Section 3.1.1).
- **taskids_list** is a queue which contains identifiers of such tasks whose descriptions have not been saved by the logging mechanism yet. This queue occurs only as an element of a *created_tasks* hash table.

- **created_tasks** is a hash table whose keys are identifiers of those tasks which are under processing on the current node (on the primary Maple kernel or on the external Maple kernels) or which have already finished their execution but which have spawned some tasks whose descriptions have not been saved by the logging mechanism yet. Every element of this table contains a *taskids_list* queue which holds the identifiers of the tasks spawned by the corresponding task. When a task has finished its execution, the scheduler does not send further its result, until the corresponding entry of this hash table contains an empty *taskids_list* queue (see Figure 4.6).

- **result_of_the_finished_tasks** is a hash table whose keys are identifiers of those tasks which have already finished, but which have spawned some tasks whose descriptions have not been saved by the logging mechanism yet. This hash table stores the results of these tasks, until the scheduler can send further their results (see Figure 4.6).

- After a thread has saved a task description, it sends a message **ack:<*t*>** to the node which created this task (where t is the identifier of the task).

New Files

- A file **sessionid** stores the identifier of the last session which used the logging mechanism. When the `dist`[`initialize`] command is executed, the system initializes the current session identifier. It checks whether a previous session identifier exists on the storage system. If it exists, it reads and increments it. If it does not exist, the current session identifier is set to 0.

- A file **link.*t*** contains the identifier of a task whose description is identical with the description of t. Namely, the system does not create a file `descr` and a file `result` more than once for the tasks whose descriptions are identical. But it creates a file `link` for each latter task, which contains identifier of the first logged task. With this file, the system is able to recover the common result and the common description of these tasks by their identifiers.

Starting Logging in Safe Mode

The user can switch on the Safe Mode by the call `dist[logging](2)`. This call may be issued only after a session is established (see the description of call `dist[initialize]` in Section 3.1.1).

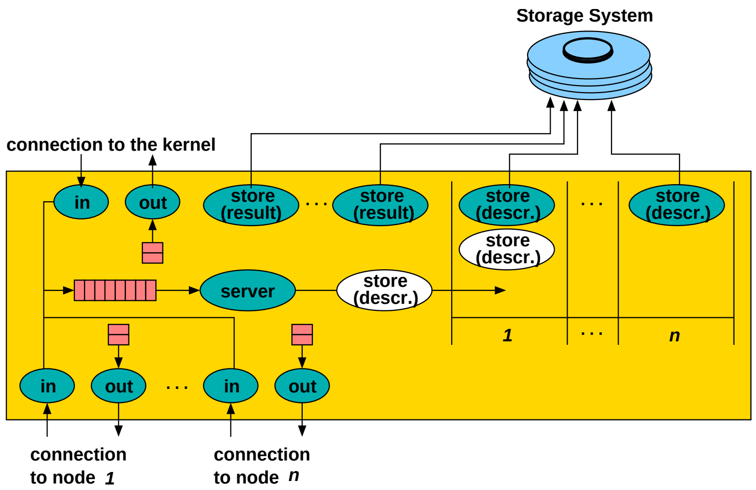

Figure 4.5: *Scheduler Architecture on the Root in Safe Mode*

The effect of this command is that the Maple frontend sends a message `stable:<2>` to the root scheduler. After the root scheduler has received this message, it creates the `logging` directory, (if it does not exist) and it saves the current session identifier into a file `sessionid`. Then it broadcasts message `stable:<2>` further to every node. By this notification, every node changes to Safe Mode.

Logging Task Descriptions in Safe Mode

When the root scheduler wants to save a task description to the storage system, it creates a concurrent thread for doing this, but it does not start the thread immediately (see Figure 4.5).

First, the root scheduler checks whether the identifier of the node which created this task occurs as a key in this hash table. If it does not occur, there is no description from this node currently being logged. Therefore, the root scheduler puts the reference of the thread with the node identifier as a key into this hash table and starts the thread.

If an entry whose key is the searched node identifier occurs in this hash table, the thread has to wait until the predecessor threads have finished their execution (see the threads denoted by white ellipses on Figure 4.5). Therefore, the root scheduler replaces the reference of the previous thread with the reference of the new thread in the corresponding hash table entry and creates a link from the previous thread to the new thread. Thus the previous thread can start the new one before it finishes its execution.

Before a thread finishes, it checks whether another thread from the same

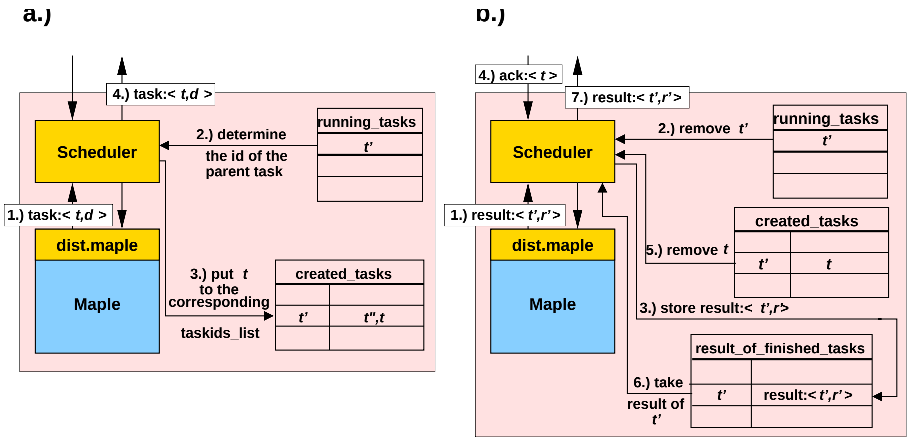

Figure 4.6: *Handling of Spawned Tasks*

node waits for a logging and, if yes, starts it. Otherwise, it removes its own entry from the *clientTable*.

Modified Task Message Handling

When a scheduler receives a message `task:`$<t,\ d>$ from a local kernel in Safe Mode, it identifies the parent task t' of task t from a list *running_tasks* (see Figure 4.6a) or from a hash table *externUsed* (depending on whether the message came from the primary Maple kernel or from an external Maple kernel). Using t' as a key, the scheduler adds t as an element to a queue *taskids_list* in the corresponding entry of a hash table *created_tasks*. Afterwards, if the current scheduler is different from the root scheduler, the message `task:`$<t,\ d>$ is sent further to the root.

After the root scheduler has received a message `task:`$<t,\ d>$ from a node (or it received it from a local kernel) and has added t to the corresponding entry of *created_tasks*, it attempts to recover the result r of t as described in Section 4.2.2. If the root can recover r , it saves the identifier t'' of the logged task (from which r was recovered) to a file `link.`t. Then root scheduler sends r in a message `result:`$<t,\ r>$ to the node which created this task and it also sends a message `ack:`$<t>$ to the same node. If the root scheduler can not recover r, it creates a concurrent thread in order to save d to the storage system (d is logged as described previously in Subsection "Logging Task Descriptions in Safe Mode"). Only after the logging of d is complete, this thread schedules the task t to a node for execution and sends a message `ack:`$<t>$ to the node which created this task.

Before a scheduler sends a task to a Maple kernel for processing, it adds a new entry to *created_tasks*. The key is the identifier of the task and the element is an empty *taskids_list*. Afterwards, the scheduler puts the task identifier to the end of *running_tasks* or puts the task with the identifier of an external Maple kernel as a key to a hash table *externUsed* (depending on whether the scheduler sends the task to the primary Maple kernel or to an external Maple kernel).

Modified Result Message Handling

When a Maple kernel sends a result r in a message `result:`$<t, r>$ to the local scheduler, this scheduler removes t from *running_tasks* (see Figure 4.6b) or removes the corresponding row from *externUsed* (depending on whether the `result` message came from the primary Maple kernel or from an external Maple kernel). Then, using t as a key, the scheduler checks whether the corresponding entry/*taskids_list* is empty in *created_tasks*. If it is empty, the system is allowed to use r and to remove this entry from the *created_tasks* hash table. If it is not empty, the system is not allowed to use r until this entry has become empty. Therefore, the scheduler puts r with t as a key to a hash table *result_of_the_finished_tasks*.

Acknowledgement Message Handling

When a scheduler receives a message `ack:`$<t>$, it takes t and starts to look for it in *created_tasks*. More precisely, the scheduler compares t with the first element of every *taskids_list* in *created_tasks* (it is enough to check the first elements, because the descriptions of tasks which are created by the same node are saved sequentially and therefore the `ack` messages also arrive sequentially). If the scheduler finds an element which equals t, it removes it from the corresponding entry/*taskids_list*. If this entry becomes empty, the scheduler removes the entry from *created_tasks* (whose key is the identifier t' of the parent task). The scheduler checks whether there exists a result r' for t' in *result_of_the_finished_tasks*. If the result exists, it removes the corresponding entries from both *created_tasks* and *result_of_the_finished_tasks* and it sends a message `result:`$<t', r'>$ to the node which created the task t' (the session begins to use r').

Modified Wait/Request Message Handling

If a scheduler on a non-root node n receives a message `wait:`$<t>$ from a local kernel, where t consists of a group $<s, n, i>$ and s less than the current session identifier (task was created in a previous session), then the scheduler sends a message `request:`$<t, n>$ to the root.

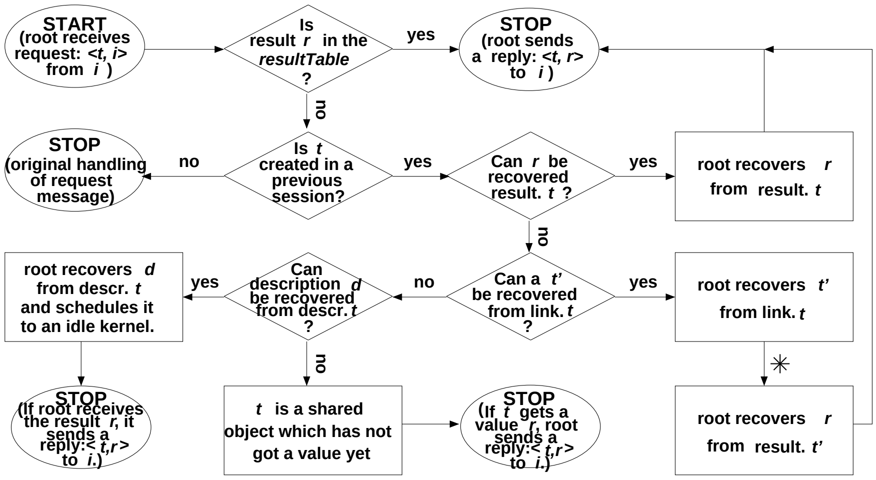

Figure 4.7: *Task recovery by task identifier*

When the root scheduler receives a message `wait:`<t> from a local kernel or a message `request:`<t, n> from another node, where t either was created on the root in this session or was created on any node in a previous session, it looks for the result r of t in a table *resultTable* (see Figure 4.7). If r is located there, the root returns it either in a message `result:`<t, r> to the corresponding local kernel or in a message `reply:`<t, r> to node n (depending on where the request came from).

If r is not in *resultTable* and t was created in a previous session, the root scheduler tries to recover the result r from a file `result.`t. If r cannot be recovered from this file, the root scheduler attempts to recover it from another file `result.`t' which is identified by task identifier t' which is the content of a file `link.`t. (* Remark for Figure 4.7: If t' can be recovered from `link.`t then the system does not need to check the existence and the correctness of `result.`t', because a link file may be created if and only if the corresponding result was already recovered successfully by its description in a previous session).

If the recovery is successful, the root scheduler sends back the result r either in a message `result:`<t, r> or in a message `reply:`<t, r> to the requester. Otherwise, the root scheduler attempts to restore the task description d from file `descr.`t. If it is successful, the root scheduler schedules d in a message `task:`<t, d> to a node for execution (the requester will get r, after the root receives it in a message `result:`<t, r>).

If it does not manage to recover either the description, the system treats t as an identifier of a shared object which has not got any value yet (if the identifier denotes a task, the program will consequently deadlock; this error cannot occur when the logging was performed in Safe Mode).

4.4 Handling of Shared Objects

As mentioned in Section 3.2.5, a shared object is nothing but a result descriptor; the only difference is that a shared object t *that is still empty* can receives its value r by a message `put:`$<t,r>$ that may be issued by any task knowing t.

4.4.1 Algorithm

For handling the shared objects, the logging and recovery mechanism is extended as follows:

Logging Since there is no task description d of a shared object, no hash file is written and no description file is created, if a `data:`$<t>$ message is issued.

If a `put:`$<t,r>$ is issued on a node different from the root, a duplicate of this message is forwarded to the root and processed analogously to the `store:`$<t,r>$ messages for task results.

Recovery If the root receives a message `request:`$<t, n>$ where t refers to a previous session and is destined for a shared object, it proceeds analogously to the description in Section 4.3.1 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 `reply:`$<t,r>$. Otherwise, the root need not do anything because the result will be computed by another task *t'*.

4.4.2 Implementation

A shared object has no description, its value is determined by a task issuing `dist`[`put`] command. The following activities which handle the shared objects are changed in the "Logging" mechanism.

Modified Put Message Handling

If the `dist[put]` command has been executed, a scheduler receives a message `put:`$<t, r>$ from a local kernel. After the scheduler processed this message, it sends a message `store:`$<t,\ r>$ to the root. When the root scheduler receives this message, it handles it as described in Section 4.2.2.

Wait Message Handling

If the `dist[get]` command has been executed, a scheduler receives a message `wait:`$<t>$ from a local kernel. In this case, the processing of the `wait` message is the same as described in case of the tasks (see Section 4.3.2).

The root scheduler cannot make a distinction between the recovery of tasks or of shared objects. This means, if it cannot restore a result of a shared object by an identifier, it tries to recover the content of the corresponding link file or description file as described in Section 4.3.2. Of course, these files exist if and only if the corresponding identifier refers to a task. So, the root does not find these files for a shared object. Therefore, the root treats the identifier as that of a shared object which has not got any value yet.

Restrictions

It is not allowed to use the `dist[clear]` command when the logging mechanism is switched on because the system is not capable to save and store a consistent state of the computations for the logging mechanism, when the value of a shared object changes.

Chapter 5

Tolerating Connection Failures

The previous chapter dealt with a mechanism by which a session restarted after a failure can reuse previously computed results. In this chapter (and in Chapter 6), we explain some mechanisms that enable a session to cope with some kinds of failures without aborting or deadlock.

These mechanisms handle connection failures which may occur in a Distributed Maple session. To detect this kind of failures, the already mentioned watchdog mechanism is applied both on the root and every node. Because of consequences of the FLP impossibility result [22] (see Section 2.2.1), this failure detector mechanism cannot make a distinction between the following situations:

- A connection breaks.
- A node crashes.
- A node is too slow in sending messages (the network latency is longer than the limited time period which the watchdog uses).

Therefore, we have to consider the possibility that any of these situations has occurred, when the failure detector reports a connection failure. Consequently, we chose the following strategy for handling failures in the system [8, 9, 10]. If the watchdog mechanism on a node detects that another node became unreachable, the percipient node handles the situation as a connection failure first. It closes the connection to this node and tries to establish a new connection to it.

Secondly, if the percipient node is the root and the reconnection is unsuccessful, then the unreachable node is regarded as a failed node and every message arrived from this node is rejected. If the percipient node is a non-root node and the reconnection is unsuccessful, then it attempts to connect to the unreachable node trough the root. Later, either the root may declare the unreachable node as failed node or the percipient node may attempt to connect to this node again.

In the rest of this chapter, we explain the two mechanisms that handle connection failures: in Section 5.1 "Reconnecting with Message Acknowledging" and in Section 5.2 "Tolerating Peer Connection Failures".

5.1 Reconnecting with Message Acknowledging

In this section, we describe the "Reconnecting" mechanism of Distributed Maple. Its task is to attempt to recreate an interrupted connection between the root and any other node and to resend lost messages. For achieving this goal, this mechanism introduces a *message acknowledging* mechanism.

In practice, the "Reconnecting" mechanism is called after the root detects that a node has become unreachable, but before the "Tolerating Non-Root Node Failures" mechanism described in Section 6.1 is applied.

5.1.1 Assumptions and Guarantees

For using the "Reconnecting" mechanism, the following conditions have to be satisfied:

- communication is reliable(we suppose messages never become corrupted or duplicated and messages are lost only in case of connection failures), and

- no Byzantine failures occur.

The benefit of the "Reconnecting" mechanism is that if the reconnection is successful then the tasks which have already been scheduled to the reconnected node do not need to be rescheduled to another node and their started (and perhaps finished) processing does not have to be repeated (unlike the "Tolerating Non-Root Node Failures" mechanism). If the reconnection is unsuccessful, the "Tolerating Non-Root Node Failures" mechanism can still be called.

In other words, a session is able to reduce the loss of resources in some kinds of failure situations by the "Reconnecting" mechanism. Furthermore, the message acknowledging mechanism introduced by this mechanism is indispensable for the accomplishment of the "Tolerating Root Failures" mechanism described in Section 6.3.

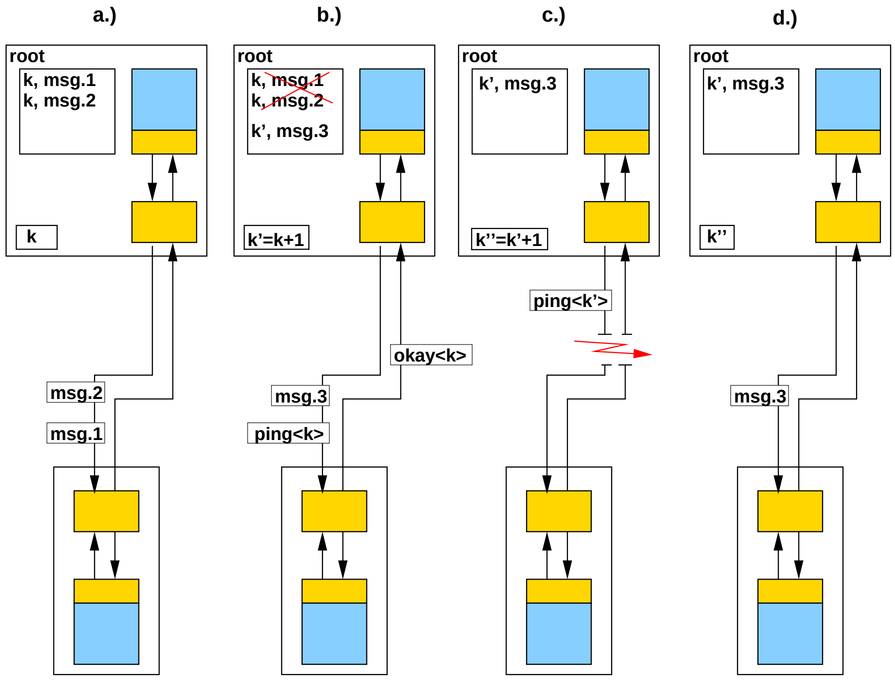

Figure 5.1: ***a.)*** *and* ***b.)****: Message Acknowledging* ***c.)****and* ***d.)****: Resending of not Acknowledged Messages after Reconnection*

5.1.2 Algorithm

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 use the watchdog mechanism (see Section 3.2.3). By this, the root checks periodically the connection to every node even as the non-root nodes check their connections to the root.

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 Non-Root Node Failures" mechanism (see Section 6.1). 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).

There is a problem that the root and the reconnected node i 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, and node i can resend some task, result, store and wait messages to the root. The root also deals with those messages which were sent through it to i by any other node. All other nodes (except the root and i) are not affected by a connection failure in case of successful reconnection.

For resending the corresponding messages, every node maintains a set M_j, which contains all messages sent to j (see Figure 5.1a). On the root, this set contains also those 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 5.1b). 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 3.2.3). 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 5.1c and d).

Since it may occur that a message reaches its destination, but it has never been acknowledged due to a connection failure, some messages may arrive twice after a reconnection. This is not a problem, because the system tolerates redundant messages.

5.1.3 Implementation

The first step for achieving the “Reconnecting” mechanism is to start a new watchdog thread on the root similarly as on the non-root nodes. By this thread, the root is able to check the reachability of non-root nodes. Furthermore, we have to take care of the following issues during the implementation:

- A non-root node has to be able to accept a connection request from the root at any time even if it has not detected any connection failure yet.
- After a successful reconnection, the set of messages which are lost due to a connection failure between the root and a node has to be determined in a session.

New Data Structures and Messages

- **PingId** is an integer variable. On every node i, this variable is assigned to each connection via which i interacts to another node and it is always initialized with 0 after the corresponding connection is established. This variable is used as a counter to generate unique identifiers for `ping` messages sent to the same node.

- The new version of the message **ping:<*index*>** has an argument *index* which uniquely identifies the message.

- Originally, `okay` messages are used for acknowledging of `ping` messages. The new version of the message **okay:<*index*>** has an argument *index*, with which the acknowledged `ping` message can be identified.

- A **MsgRegistry** is a hash table whose keys are numbers which identify `ping` messages and whose elements are lists of messages. On a node i, such a hash table is assigned to each connection via which i interacts to another node.

- A message **reconnection:<*i*>** can be sent only locally to the server thread either by the watchdog thread on the root or by the connect thread on a non-root node. The server thread on the root receives such a message if and only if the watchdog thread detects that a node has become unreachable. A server thread of a non-root node receives this message if and only if the root has already reconnected to it The argument i is the identifier of the unreachable node.

Initialization of Data Structures on a Root

During a session initialization, an empty hash table *MsgRegistry* is created to each connected node.

A watchdog thread is started if and only if this mechanism is activated. This can be done by the call `dist[logging](4)` (see Section 7.1).

Message Acknowledging

As mentioned in Section 3.2.3, the couple of `ping` and `okay` messages are originally used by the watchdog mechanism. A message `ping` is sent if any message has not arrived from a connected node for a limited period of time. This message is always acknowledged with a message `okay` by a connected node. Since the sequence of the messages sent via a connection cannot be changed, the `ping` and `okay` messages can be used for acknowledging received messages as follows.

For this, a variable *pingId* and a hash table *MsgRegistry* is assigned on node i to each connection via which i interacts with another node j. Before a message (different from `ping` and `okay`) sent to j, the current value of *pingId* is used as a key to store the message in *MsgRegistry* (see Figure 5.1a).

A message `ping:`<*index*> is sent to j if either any message has not been recently received from j (original purpose) or the number of messages in

MsgRegistry reaches an upper limit. The value of its argument *index* comes also from the current value of *pingId*, but then *pingId* is incremented (see Figure 5.1b).

If node j receives a message `ping:`<*index*> from i, j simply returns *index* in a message `okay:`<*index*> to i. When this `okay` message arrives at i, the list of messages which is stored with *index* as a key in *MsgRegistry* is removed (see Figure 5.1b).

In case of a successful reconnection, it is enough to resend those messages which are contained by the corresponding *MsgRegistry* (i.e., which have not been acknowledged by the reconnected node, see Figure 5.1 c and d).

Modified Watchdog Thread

If a *watchdog thread* on a non-root node detects that the root became unreachable, it waits for a connection request from the root for a limited time period. If the request does not arrive, the node aborts.

If the new watchdog thread on the root detects that a node i became unreachable, it sends a message `reconnection`<i> to the server thread.

Modified Connect Thread

If a *connect thread* of a node receives a connection request from the root, it sends a message `reconnection`<*rootId*> to the server thread of the node. Its argument *rootId* denotes the identifier of the root.

Reconnection Message Handling

When the server thread of the root receives a message `reconnection:`<i>, it first informs the user about the failure. Then it attempts to recreate the connection to node i in the same way in which a non-root node creates a peer connection to another node during normal operation. If i does not respond to this connection request, the root performs “Tolerating Non-Root Node Failures” mechanism on i (see Section 6.1). If the connection to i is reestablished, the root increments the corresponding *pingId*. Then, it takes and deletes all messages from the corresponding *MsgRegistry* and resends them to i.

If a server thread of a non-root node receives a message `reconnection:`<*rootId*>, it simply resends the non-acknowledged messages to the root, similarly to the root.

5.2 Tolerating Peer Connection Failures

The connection between two non-root nodes is called peer connection. Such connections are established on demand during the execution. By the "Tolerating Peer Connection Failures" mechanism, a session is capable to cope with peer connection failures without overall failure. 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.

This mechanism uses the same message acknowledging mechanism for resending lost messages as the "Reconnecting" mechanism (see Section 5.1).

5.2.1 Assumptions and Guarantees

The "Tolerating Peer Connection Failures" mechanism requires the following assumptions:

- communication is reliable (we suppose messages never become corrupted or duplicated and messages are lost only in case of connection failures),
- no Byzantine failures occur, and
- either connections never fail between root and non-root nodes or the "Tolerating Non-Root Node Failure" mechanism is switched on (see Section 6.1).

In practice, the "Tolerating Peer Connection Failures" mechanism is never used without the "Tolerating Non-Root Node Failures" mechanism. Therefore, we must take over an additional assumption, which is still required for the working of this other mechanism (see Section 6.1.1). This supposes that the root node has an ability to access to a reliable storage system.

If the system satisfies these conditions, the "Tolerating Peer Connection Failures" mechanism guarantees: if some peer connections between non-root nodes break, then the system is able continue normal operation without any deadlock situations (by resending of the corresponding messages through the root).

5.2.2 Algorithm

The watchdog mechanism located on every non-root node is also used to check regularly the peer connection between these nodes. If a node i detects that another node j became unreachable to i via a peer connection, i attempts

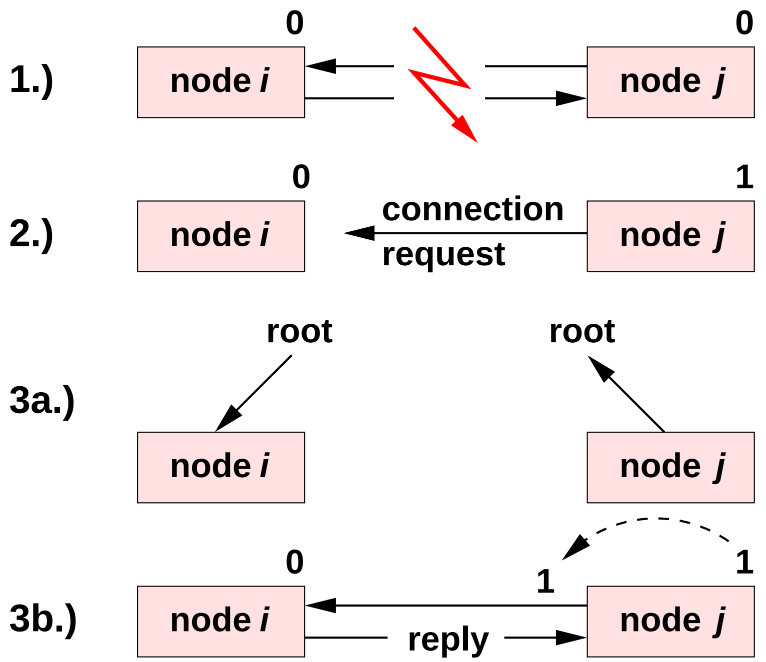

Figure 5.2: *Handling of Peer Connection Failure*

to establish a new peer connection to j in order to resend the messages to j, which might be lost (see Figure 5.2). To determine the set of such messages, the same message acknowledging mechanism is used like in "Reconnecting" mechanism (see Section 5.1.2).

- If j does not respond to the connection request of i in a certain time, all messages which must be delivered from i to j are sent through the root for a certain time (then i may attempt to contact j again).

- If j responds to the connection request of i in time, i sends to j the number of detected connection failures between itself and j. If this value is greater than the number of connection failure detected by j between j and i, then j recognizes that j has not detected a connection failure yet. Hence, j also resends the not acknowledged message to i.

If nodes on both sides of a peer connection observe a failure of the connection roughly at the same time (and both of them tries to connect to the other one), then it may occur that two pieces of one way connections are established.

5.2.3 Implementation

The "Tolerating Peer Connection Failures" mechanism is based on an extended version of the watchdog mechanism originally available in Distributed Maple (see Section 3.2.3). Namely, the non-root nodes check the connections between each other with the same strategy in this new watchdog version as they have already checked their connection to the root in the previous version, too.

In the course of the implementation of the "Tolerating Peer Connection Failures" mechanism, we took care of the following cases:

- In contrast to the connections between root and non-root nodes, the peer connections are established during the execution, if they are needed. Therefore, the system has to be able to check, whether a connection is created successfully.
- If the establishing of a peer connection was unsuccessful, the corresponding two non-root nodes have to interact via the root.
- If the establishing of a peer connection was unsuccessful, the corresponding non-root nodes may try it again after some time.
- If a peer connection breaks, both non-root nodes have to become aware of this.
- The non-root nodes have to determine which messages might be lost due to a peer connection failure.

Furthermore, we had to investigate the situation in which a node fails but a non-root detects this event earlier than the root and it tries to resend the lost messages through the root. In that case, if the root does not detects the node failure before it receives these messages it forwards them. Otherwise, it simply does not deal with them. Of course these message are lost (again) in both cases. Eventually the root detects the node failure and among others these lost messages are handled and resent again as described in Section 6.1. Thus some redundancy may occur in such a situation, because some messages might be resend twice.

New Data Structures

- **peerConnections** is a bit set, which is located in every non-root node. Each bit in this data structure belongs to a possible peer connection to another non-root node. If a peer connection is created between two non-root nodes, then the corresponding bit is set on both nodes. If a non-root node i observes that a peer connection is broken, the corresponding bit is cleared in *peerConnections* on i.

- **currentPeers** is a bit set, which is located in every non-root node. Each bit in this data structure belongs to a possible peer connection to another non-root node. From time to time, its content is synchronized to the content of *peerConnections*. A non-root nodes checks only those existing peer connections by watchdog mechanism whose bits are set in *currentPeers*. If a non-root node i detects that a peer connection broke, the corresponding bit is cleared in *currentPeers* on i.

- **status** is a hash table whose keys are non-root node identifiers and whose elements are integer values. If such a value is 0, messages may be sent directly to the corresponding non-root node via peer connection. If such a integer value is greater than 0, messages are sent to the corresponding non-root node through root.

- **numberOfFailures** is a hash table whose keys are non-root node identifiers and whose elements are integer values which show how many times the peer connection to the corresponding non-root node broke. If a non-root node observes that another non-root node has become unreachable via peer connection, the non-root node increments the corresponding value in this data structure. When a new peer connection is established, the corresponding values of these two non-root nodes are compared. If they are not equal, one of non-root nodes does not observe the peer connection failure between this two nodes.

- A message **brokenpeer:<*i*>** is sent if a non-root node observes that one of non-root nodes became unreachable via peer connection. The argument i is the identifier of the unreachable non-root node.

- **sockets** is a hash table whose keys are non-root node identifiers and whose elements are references to those sockets via which a node interacts with other nodes.

- **inThreads** is a hash table whose keys are non-root node identifiers and whose elements are references of input threads which handle the arriving messages from the corresponding non-root nodes.

- A message **outclose** is sent if a node intends to stop one of its output threads. The corresponding thread does not deliver this message, but it finishes its own execution.

Activating of Tolerating Peer Connection Failures

The “Tolerating Peer Connection Failures” mechanism can be switched on together with the “Tolerating Non-Root Node Failures” mechanism by the call `dist[logging](3)`.

If this mechanism is activated, a variable *pingId* and a hash table *MsgRegistry* is initialized for all established peer connections and the messages sent via such connections is acknowledged with the message pair `ping:`<*index*> and `okay:`<*index*> even as in the “Reconnecting” mechanism (see the description of *pingId*, *MsgRegistry* and message acknowledging in Section 5.1.3).

Modified Message Sending Protocol between Nodes

When a non-root node i wants to send a message to another non-root node j, i tries to sent it directly via a peer connection. If such a connection has not existed between i and j yet and the value referred by j in the hash table *status* is equal to 0, then i sends a connection request to an announced communication port of j, on which the *connect thread* of j is listening (see Figure 5.2).

- If the peer connection is established successfully between i and j, both nodes exchange their identifiers between each other first in order to identify themselves.

  Then i, which issued the connection request sends to j the corresponding value from the hash table *numberOfFailures* located on i. The connect thread on j compares this received value with another one referred by i in the hash table *numberOfFailures* located on j. If the received value is the greater one (which means a previous peer connection between i and j failed, but j has not detected it yet unlike i), then the *Closing_Peer* operation is called with i as an argument by j.

  Afterwards, both nodes set the corresponding bit in their own bit sets *peerConnections*. At last, they start input threads and output threads belonging to the created peer connection and they put references to the related socket and to the newly started input threads to their hash tables *sockets* and *inThreads*. A message can be sent via this peer connection only after this procedure.

- If node j does not respond to the connection request of i during a limited time, i sends the message to j through the root and it sets the corresponding value of the hash table *status* to a positive number. This implies that every message sent from i to j will be delivered through the root until the value referred by j as a key in *status* become 0 again.

- If the value referred by j in *status* is greater than 0, i simply sends the message to j through the root

Modified Watchdog Thread

The new watchdog thread of non-root nodes extends the activity of the original watchdog, such that it periodically checks an existing peer connection if the bit assigned to this connection in the hash table *currentPeers* is set. Namely, the action of a watchdog thread is split to time periods. When such a time period starts, the watchdog thread synchronizes the content of *currentPeers* to the content of *peerConnections*. Hence, during a time period, the watchdog thread waits for messages via only those peer connections (or forces a reply via them at least) which were established in a previous watchdog period (in other words, each connected non-root node has at least one such a time period to send some message via a new created peer connection).

If a watchdog thread detects that a non-root node j became unreachable via peer connection, it simply calls the Closing_Peer operation with j as an argument.

After the checking of all peer connections in a watchdog time period, the watchdog thread decrements all those values contained in *status* which are greater than 0.

Closing_Peer Operation

This operation is called with an argument j after a peer connection failure either by the watchdog thread if it detects that a non-root node j has become unreachable or by the connect thread due to a connection request from j. To avoid to apply this operation to the same broken connection twice (which may cause deadlock), the value of the bit j in bit set *peerConnections* is investigated first. If the checked bit is cleared, the *Closing_Peer* operation returns. Otherwise, it clears the checked bit in *peerConnections* and the same bit in *currentPeers*, too (the Closing_Peer operation is encapsulated and it can be invoked by only one thread at the same time).

Then it increments the corresponding value referred by j in *numberOfFailures*. It closes the communication channel to j by using the corresponding references stored in hash tables *sockets* and *inThreads* and by sending a message `outclose` to the corresponding output thread. Finally, this operation sends a message `brokenpeer:`<j> to the server thread.

Brokenpeer Message Handling

When a non-root node i receives a message `brokenpeer:`$<j>$, it forwards this message to the root and resends to j those messages from the corresponding *MsgRegistry* which have not been acknowledged yet (like in the case of "Reconnecting" mechanism, see Section 5.1.3).

When the server thread of the root receives a message `brokenpeer:`$<j>$ from a node i, it informs the user of that i cannot interact with j via peer connection.

Chapter 6

Tolerating Node Failures

In this chapter, we introduce some fault tolerance mechanisms to avoid a session failure in case of the failure of any node (even if the root). Furthermore, we present a simple algorithm by which a session is able to restart failed nodes in order to reduce the loss of resources.

Lots of systems [5, 37, 36] which provide automatic restart and recovery in response to node failures use *checkpointing*. This is a generally usable and stable algorithm, but it is not efficient enough (e.g.: the execution phase is periodically interrupted by checkpointing phase in normal operation, the whole session has to return to the last checkpoint after a node failure, etc). Furthermore, this mechanism is not able to tolerate server (or root) failure without overall failure.

One possibility to achieve tolerating server (or root) failure without session failure is to apply some kind of *leader election* algorithm. If the server fails, one of the other session members is elected as the new server. The problem with such approaches is that most leader election algorithms were developed only for synchronous distributed system or had assumptions that were too strong for real-life applications [41] (e.g.: that links between nodes never fail). If the new server is selected, a distributed session can be restarted from a consistent state.

To our knowledge, the *Invitation Algorithm* made by Garcia-Molna for asynchronous distributed systems [25] comes closest to our research (for more details see Section 2.2.3). But this algorithm still cannot handle the situation in which a session/network is split to two or more parts. In such a case, a group of the nodes which lost the connection with the original server triggers a server election (there may exist more than one server at the same time). Moreover, an analysis of the Invitation Algorithm [69] shows that its specification is undesirably strong: in some situations, it requires that a connection between two nodes never fails.

In Distributed Maple, the applied functional task model helps a lot for simplifying the problem of node failures, because we have to restart only those tasks which were under processing on the failed node [64, 65, 8] (and of course resend the lost messages). Hence, the whole session does not have to interrupt its execution and can return to a stored consistent state of it.

The case of root failures is not so simple. We also developed a leader election algorithm to choose a new root from the pool of non-root nodes after a root failure was detected [9, 10]. However, in contrast to the Invitation Algorithm, no situation may occur in which there exist more than one root at the same time. If the new root is elected, the features of the functional task model and of the "Logging" mechanism facilitate to set a consistent state of the new root (without stopping the whole execution and returning to a stored state of the session).

In the rest of this chapter, we present and describe three mechanisms: in Section 6.1 "Tolerating Non-Root Node Failures", in Section 6.2 "Restarting after Node Failures", and in Section 6.3 "Tolerating Root Failures".

6.1 Tolerating Non-Root Node Failures

In this section, we sketch a mechanism that enables a session to cope with non-root node failures without aborting (the root continues operation with the remaining nodes).

6.1.1 Assumptions and Guarantees

For using the "Tolerating Non-Root Node Failures" mechanism, we assume that the Safe Mode of "Logging" mechanism described in Section 4.3 is switched on and works correctly. Namely, if a non-root node becomes unreachable, the root restores the corresponding logged task descriptions from the storage system in order to reschedule them to other nodes. (The Safe Mode guarantees, that every task description is saved to the storage system, before it is scheduled.)

This means, we also use all the assumptions here which we have already for the proper functioning of logging mechanism in Section 4.1:

- access to a reliable storage system,
- reliable communication (we suppose messages never become corrupted or duplicated, but we do allow to lose messages in case of connection break),

- no Byzantine failures.

If the system meets these conditions, this mechanism guarantees:

- all guarantees of the Safe Mode and
- even if some non-root nodes fail or some connections between the root and a non-root node break permanently, then the system is able to continue normal operation (by rescheduling of the corresponding tasks and resending of the corresponding messages).

Furthermore, we must also mention that the application of the functional task model in the system permits the continuous processing of scheduled tasks on faultless nodes even if a node failure is detected and tolerated in a session.

6.1.2 Algorithm

As mentioned in Section 5.1, the root regularly monitors the connection to each node by watchdog mechanism. If a node becomes unreachable to the root and the root cannot contact to it by the "Reconnecting" mechanism, then this mechanism is started.

First of all, the "Tolerating Non-Root Node Failures" mechanism declares the unreachable node *dead* (see Figure 6.1). 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 descriptions 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 Section 4, 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:

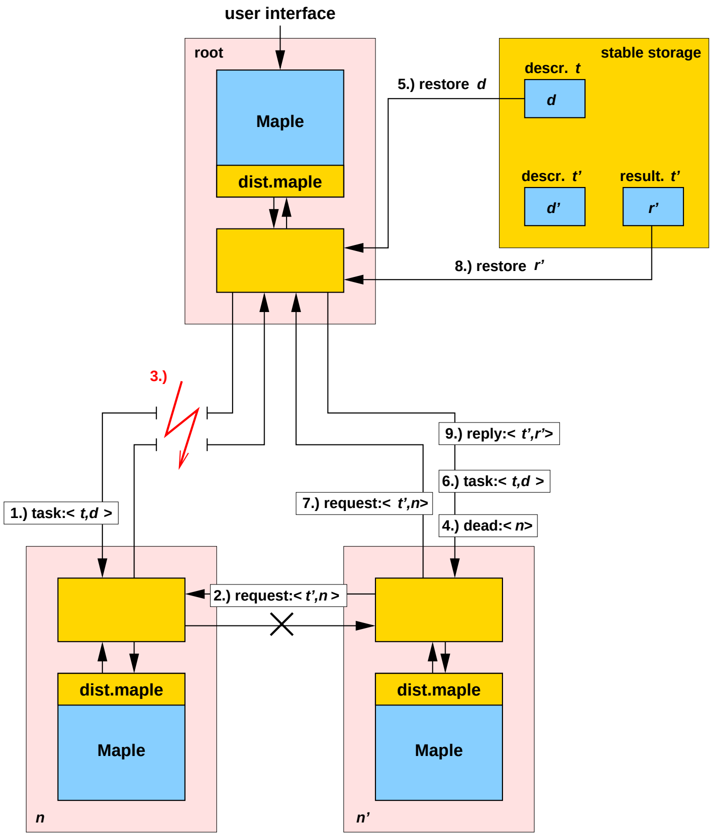

Figure 6.1: *Tolerating Non-Root Node Failures*

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 $T_n - T_n^r$ 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 descriptions are stored on n; for 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:`$<t,n'>$ messages sent to node n but not yet answered by a `reply:`$<t,r>$. When n is marked dead, the node resends all messages in R_n to the root which will eventually answer them.

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.

6.1.3 Implementation

The "Tolerating Non-Root Node Failures" mechanism is based on the watchdog mechanism originally available in Distributed Maple. This means, the root checks periodically all nodes. If a non-root node does not send any message to the root during a limited time period, the root thinks, the node became unreachable. In the course of design of this mechanism, we took into consideration that watchdog mechanism cannot make a distinction between the following situations (as discussed in the beginning of Chapter 5):

- A non-root node crashes.
- The connection between the root and a non-root node breaks.
- The answer from a non-root node arrives too late (the network latency is longer than the limited time period which the watchdog uses).

If the root interrupts the connection to a node (maybe, it tries first to reconnect to the node by "Reconnecting" mechanism described in Section 5.1, if this mechanism is activated), but the node is still alive, the node will abort after some time.

When the root observes that a node became unreachable, it reschedules those tasks which were scheduled on the node and sends a broadcast to all other nodes. Due to this broadcast, all nodes (the root and the others) resend those messages to the root which might be lost. Namely, the task results which were located on the unreachable node are restored and provided by the root.

New Data Structures and Messages

- **crash** is a hash table whose keys are non-root node identifiers and whose elements are boolean values which show the states of a non-root node. If a value is false, the corresponding node lives. If a value is true, the corresponding node became unreachable.
- **sockets** is a hash table whose keys are non-root node identifiers and whose elements are references to those sockets via which the root or a node interacts with other nodes.

- **inThreads** is a hash table whose keys are non-root node identifiers and whose elements are references of input threads which handle the arriving messages from the corresponding non-root nodes.
- A message **outclose** is sent if a node intends to stop one of its output threads. The corresponding thread does not deliver this message, but it finishes its own execution.
- **scheduled_tasks** is a hash table whose keys are node identifiers and whose elements are lists of task identifiers which are scheduled for processing to the corresponding node. This data structure is located only in the root.
- A message **dead:**$<\boldsymbol{i}>$ is sent if the watchdog of the root observes that one of nodes became unreachable. The argument i is the identifier of the unreachable node.

Initialization of Data Structures on a Root

During a session initialization, the root puts for each newly established connection the references to a socket and to the corresponding input thread to a hash tables *sockets* and *inThreads*. It also inserts a value "false" with an identifier of each connected node as a key into a hash table *crash*.

An empty hash table *scheduled_tasks* is created and a watchdog thread is started if and only if this mechanism is activated. This can be done by the call `dist[logging](3)` (see Section 7.1).

Initialization of Data Structures on Nodes

A hash table *crash* is filled up with value "false" for every non-root node on each non-root node at the time of the session initialization.

As written in Section 3.2.2, if a non-root node intends to send a message to another non-root node via a peer connection, but such a connection does not exist between these two nodes yet, then it is created. After such a connection has been established, the nodes on both sides put references to the related socket and to the corresponding input thread to the hash tables *sockets* and *inThreads*.

Modified Task Scheduling

Before the root schedules a task in a message `task:`$<t,\ d>$ to a node i, it puts the task identifier t with node identifier i as a key to a hash table *scheduled_tasks* (see Figure 6.2a).

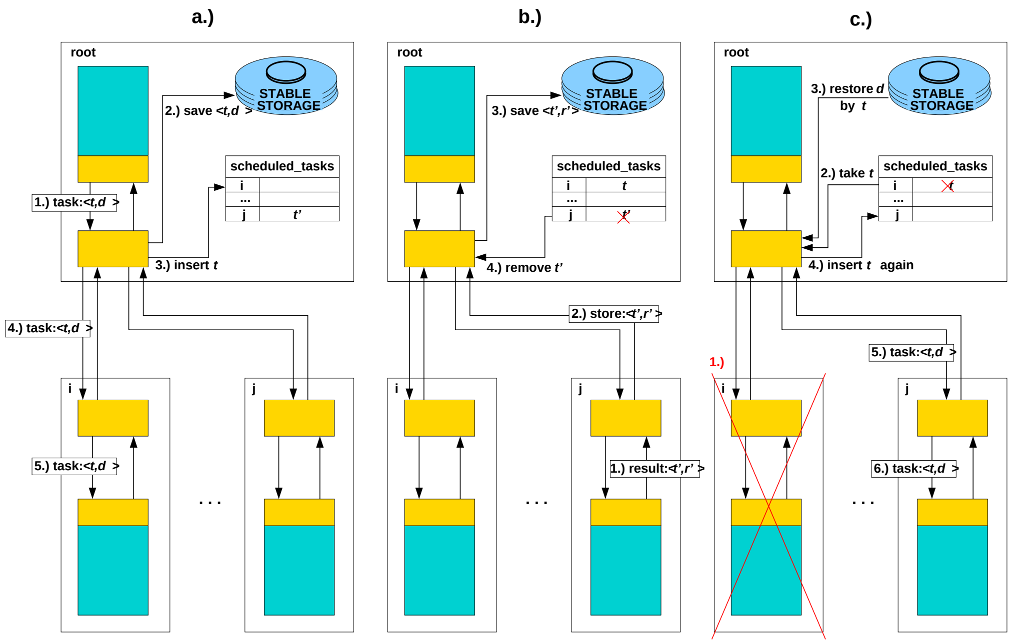

Figure 6.2: ***a.)** Task Scheduling* ***b.)** Result Logging* ***c.)** Rescheduling*

If the root receives a message `store`:<*t'*, *r'*> from a node *j*, it logs the received result *r'* and removes the task identifier *t'* from hash table *scheduled_tasks* by using *j* as a key (see Figure 6.2b). If the root does not find *t'* in the corresponding row of hash table *scheduled_tasks*, then *t'* denotes a shared object.

Then the root checks whether any received message `wait`:<*t'*> or message `request`:<*t'*, *n*> has already waited for the result *r'*. If such a request has arrived, it sends back *r'* (namely, if the node which emitted the task *t'* fails, then all requests related to *r'* are forwarded to the root).

Dead Message Handling

By the *watchdog thread* of the root, the root monitors the nodes. If this thread detects that a node *i* became unreachable, it sends a message `dead`:<*i*> to the server thread of the root (but only after an unsuccessful reconnection to *i*, if the "Reconnecting" mechanism has been activated, see Section 5.1).

When the server thread of the root receives a message `dead`:<*i*>, it first informs the user about the failure. Then it closes the communication channel to *i* by using the corresponding references stored in hash tables *sockets* and *inThreads* and by sending a message `outclose` to the corresponding output thread. Then it adjusts the corresponding line in hash table *crash* to a value "true" by using i as a key and it broadcasts message `dead:`$<i>$ to all reachable nodes.

Afterwards, the root checks in hash table *scheduled_tasks* whether any task was under processing on node i, restores the descriptions of such tasks from the storage system and reschedules them (see Figure 6.2c).

Finally, it looks through stack *externStack* and table *externWaits* whether there exists any unanswered request which was forwarded to node i and sends them to itself (on every node, these two data structures contain the wait messages received from the local primary and additional Maple kernels).

If a non-root node receives a message `dead:`$<i>$, it checks whether it has a peer connection to node i. If yes, it closes the socket of this connection and stops the input threads belonging to it by using the references in hash tables *sockets* and *inThreads* (the corresponding output thread is also stopped by sending a message `outclose`). It also put a value "true" with i as a key to hash table *crash*. Finally, similarly to the root, the node resends the unanswered wait messages to the root which were sent to node i.

Modified Communication Protocol

Before the root broadcasts a message, it checks in hash table *crash*, which nodes live and sends the broadcast message only to them.

Before sending a result message to a non-root node, it is checked whether the node is reported as unreachable. In this case, the message is not sent.

If a node wants to send a request message to an unreachable node, then it sends it to the root. When the root receives a request message or a wait message with a task identifier created by an unreachable node, it tries to restore its result from the storage system. If this is not possible, the root will eventually receive an answer by a store message and then send an answer.

6.2 Restarting after Node Failures

We have implemented a quite simple mechanism by which the root may attempt to restart the crashed or aborted nodes.

6.2.1 Assumptions and Guarantees

It does not make sense to use the "Restarting" mechanism without the "Tolerating Non-Root Node Failures" mechanism. Therefore, we reckon the "Restarting" mechanism as some kind of extension of the "Tolerating Non-Root Node Failures" mechanism.

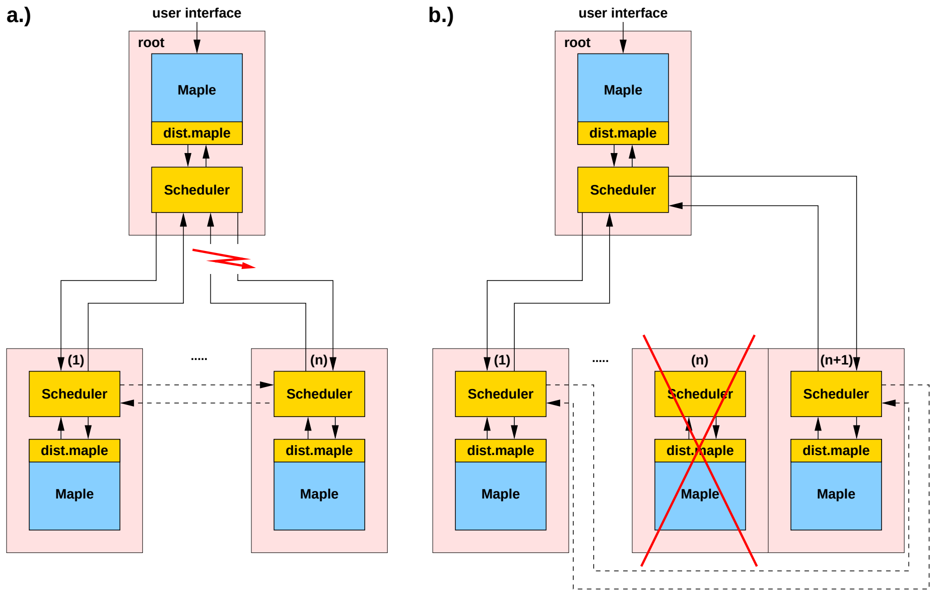

Figure 6.3: ***a.)*** *A node becomes unreachable to the root.* ***b.)*** *The root restarts the node with new identifier. If the unreachable node still alive, it eventually aborts.*

Accordingly, we can use the same assumptions for this mechanism which we use for "Tolerating Non-Root Node Failures" mechanism described in Section 6.1.1. No more assumptions are needed (nevertheless, if we want to use the "Restarting" mechanism together with the "Tolerating Root Failures" mechanism, then an additional restriction has to be introduced, see in Section 6.3.1). The benefit of the "Restarting" mechanism is that it minimizes the loss of resources (nodes) during an execution.

6.2.2 Algorithm

After the unsuccessful reconnection to node i, the "Tolerating Non-Root 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 (see Figure 6.3). 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 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 Non-Root Node Failures" mechanism). i eventually aborts.

6.2.3 Implementation

The central element of this mechanism is a concurrent thread on the root, called *restarter thread*. If a node is designated as dead, this thread periodically contacts to the server thread to attempt to start a new node instead of the unreachable one on the same machine. To achieve this, we had to take care of the following issues:

- If processes of a disconnected node (which was designated as dead) and a newly started one run on a machine at the same time, then all other node must not contact the dead node instead of the new one.

- After a node has been restarted, there are two possibilities to continue normal operation. The first is that the last consistent state of the node is restored effectively to the node (this implies the reloading of those task results to the restarted node which were stored on this node earlier and which are served currently by the root). The other way is that the tasks created newly on the restarted node are distinguished (and handled separately) from those tasks that were created also on this node before its failure (these tasks have already been migrated to the root). We chose the second solution.

- The activities of the server thread and restarter thread have to be synchronized. Otherwise, it may occur that the restarter thread triggers the restarting of a node, before the previous restarting operation of that node (successfully or unsuccessfully) has finished. In such a case, more than one node may started with the same node identifier on the same machine.

A simply and effective solution for the first two issues as mentioned already is to give a new and unique identifier for the restarted node (or in other words, starting and initialization another node instead of the failed one on the same machine).

New Data Structures

- **starting_flags** is a hash table whose keys are identifiers of dead nodes and whose elements are boolean values. If a value "true" means a restarting attempt of a node has already been triggered, but not finished. By this data structure, a kind of synchronization is achieved between the server thread and the restarter thread.

Activating Restarting Mechanism

The user can switch on the "Restarting after Node Failures" mechanism together with the "Reconnecting" mechanism by the call `dist[logging](4)`.

Insert Operation

After a node i is designated as dead, the server thread on the root calls the Insert Operation of "Restarting" mechanism. This puts a value "false" with i as a key to a hash table *starting_flags* and checks whether the restarter thread has already been started. If it does not run yet, this operation starts it.

The Restarting

If the restarter thread is started, it waits for a bounded time period. Then it selects those keys (or node identifiers) from the hash table *starting_flags* whose values are "false". By using these selected identifiers as keys, it looks for a machine name (or an IP address) and a system type for each of them from hash tables *name* and *system* (in Distributed Maple, among others these two data structures are used for storing session initialization information).

The restarter thread sends these tuples of machine names and system types in messages `start:`<*name, system*> to the server thread (in Distributed Maple, start messages are originally used only for establishing and initialization nodes at session initialization phase). Finally it sets the referred values of the selected node identifiers to "true" in hash table *starting_flags* and it starts to repeat its activity from the beginning.

The server thread handles start messages received from restarter thread in the same way as at session initialization, except some minor changes:

- If the server thread received a start message from the restarter thread, but reestablishing of a node i is unsuccessful, then it adjust the referred value of i to a "false" in hash table *starting_flags*.

- If the server thread received a start message from the restarter thread, and reestablishing of a node is successful (this operation guarantees that the (re)started node gets a new and unique identifier), then it informs the user about the successful restarting and calls the Remove Operation of the "Restarting" mechanism. At last, it activates the corresponding fault tolerance mechanisms on the restarted node by sending a message `stable:`<*mode*> and it initializes the restarted node by sending every message `all:`<*command*> issued already in the session (in Distributed Maple, these `all` messages are stored in a list *allMessages* on every node and they are originally used for initialization of additionally created Maple kernels).

Remove Operation

After a node i has been restarted successfully, an operation is started, which removes the corresponding line from the hash table *starting_flags* by using i as a key. If *starting_flags* becomes empty, this operation stops the restarter thread.

6.3 Tolerating Root Failures

This mechanism is able to change the root node such that a session may tolerate the failure of the root without overall failure. We achieved this root fault tolerance by developing and applying a new *leader election* algorithm for our asynchronous distributed system. In this algorithm, one or more distinguished nodes with some special properties (in a general case, all nodes may have such properties in a session) are able to substitute for the root.

6.3.1 Assumptions and Guarantees

The "Tolerating Root Failures" mechanism is based on all already mentioned fault tolerance mechanisms ("Logging", "Tolerating Non-Root Node Failures", "Tolerating Peer Connection Failures" and "Reconnecting") except "Restarting after Node Failures". Therefore, the same assumptions are made as were described for the previous mechanisms: a reliable and stable storage system, reliable communication, no *Byzantine* failures and only *Stop* failures may occur. 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 6.4a) or the connection between the root and the node broke (see Figure 6.4b, 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

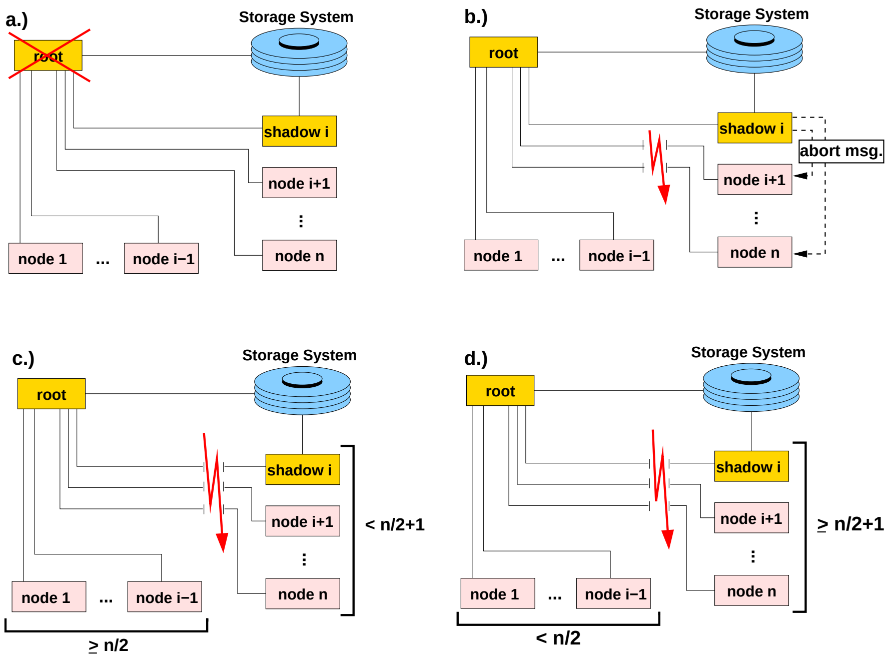

Figure 6.4: ***a.)*** *The root crashes.* ***b.)*** *The connections between the root and some non-shadow nodes break* ***c.)*** *The network is split. The root is connected to at least n/2 pieces of non-root nodes.* ***d.)*** *The network is split. The shadow node is connected to at least n/2 pieces of non-root nodes.*

6.4c); 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 6.4d), where n is the initial number of the non-root nodes.

It is possible to use the "Restarting after Node Failures" 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 6.1) 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).

If the system satisfies the conditions and restrictions mentioned above, we guarantee the following:

- If the root node crashes, then the system is able to continue normal operation such that a non-root node becomes the new root (see Figure 6.4a).

- If less than the half of number of non-root nodes become unreachable to the root, the system is able to continue normal operation. The lost nodes may be restarted later by the "Restarting after Node Failures" (see Figure 6.4c).

- If more than the half of number of non-root nodes disconnect permanently from the root and there is at least such a node among these nodes that has access to the storage system, then the system is able to continue normal operation such that a non-root node becomes the new root. If the previous root is still alive, it eventually aborts (see Figure 6.4d).
- If the system/the network is split to more than two almost equal parts, where neither the root nor one of the shadow roots can connect enough nodes, then each node aborts.

Furthermore, we would like to emphasize one more advantageous feature of our root fault tolerance mechanism. Namely, the processing of scheduled tasks on faultless nodes is never interrupted or suspended due to a root failure and a root election. It can be possible, because the parallel programming model of the system is based on functional tasks.

6.3.2 Algorithm

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 explanation assumes first that there is only one such shadow. In the last part of this section, we will generalize the mechanism to an arbitrary number of (potential) shadows.

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 6.4b). 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 6.4 a, c and d.

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 6.4c). 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 6.4a and d. In case depicted in Figure 6.4d, the old root eventually realizes that less than $n/2$ nodes are connected and aborts.

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, they resend those wait messages to the new root which might be lost, see Section 6.1). 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 table *scheduled_tasks* 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 Non-Root Node Failures” mechanism in Section 6.1).

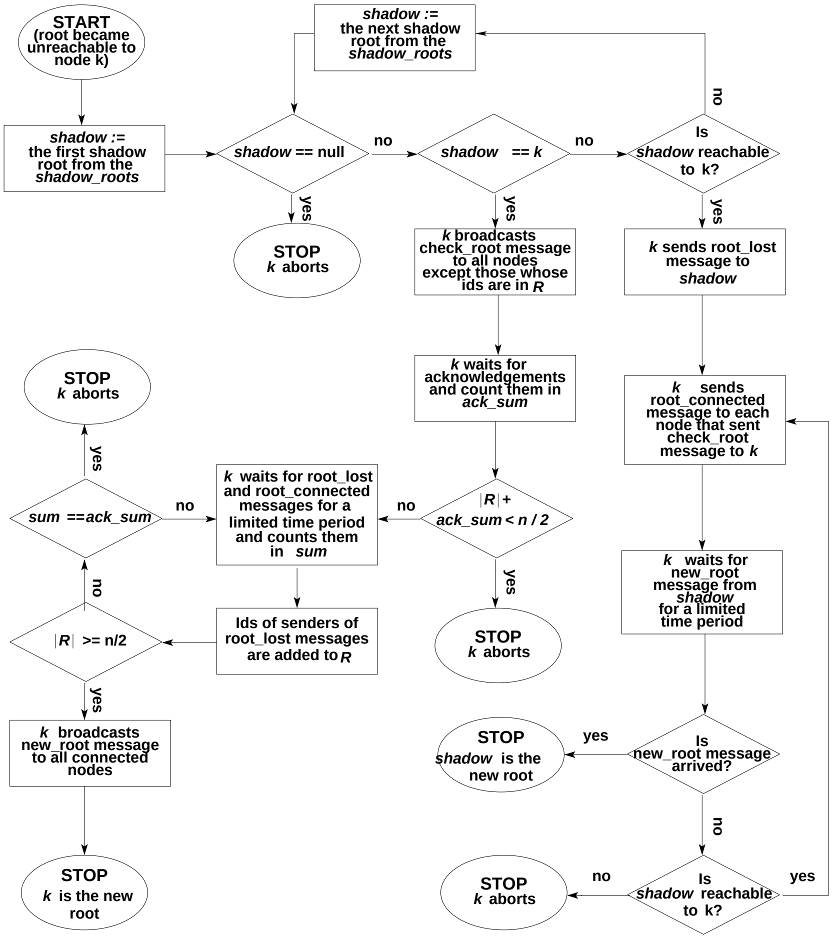

Figure 6.5: *The generalized root election method on node* k *where* n *is the initial number of the non_root nodes.* R *is a set which consists of the identifiers of those nodes that sent root_lost message to* k. |R| *signs the size of* R.

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 table *scheduled_tasks*. 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 3.2a). Therefore, after the shadow root has become the root and its initialization is finished (all lost message are resent and all lost tasks are rescheduled), the new root investigates the content of table *scheduled_tasks*. If it is empty (which means all non-root nodes are idle) and if there is no any scheduled task on the root either, then the system terminates. Otherwise the execution continuous, but the checking of this condition is repeated every time when a `store:`$<t, r>$ arrives. In a later session, the system is able to recover the result of the whole computation by the "Logging" mechanism.

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. The order of shadow nodes in *shadow_roots* is fixed.

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 6.5). 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.

6.3.3 Implementation

As mentioned in Section 6.3.1, all already described fault tolerance mechanisms (except the "Restarting after Node Failures") have to be activated for the "Tolerating Root Failures" mechanism. There are two reasons for it. The first reason is that the guarantees of these mechanisms are required for the working of our root fault tolerance mechanism. The other reason is that the using of many data structures introduced by the already presented mechanisms are indispensable for the "Tolerated Root Failures" mechanism. Therefore, many data structures whose descriptions are located in previous sections and which are established for other purposes occur frequently in this Section.

During the implementation of this mechanism, we took care of the following issues:

- Since the watchdog mechanism cannot make a distinction between a node failure and a connection failure, the situation in which a network is split and the original root and a new elected root may exist at the same time has to be avoided.

- Since Distributed Maple is strongly centralized, the root stores some extra information collected from all other components of a session. This information, which is lost due to the root failure, but which is required for the further execution of the session, has to be recollected efficiently on the elected root.

- The stable storage system has to be independent from the root and the main task of a Distributed Maple program has to be logged in it.

New Data Structures and Messages

- **initRootId** is a constant and its value is the identifier of the original root in a session. During an execution, the original root is the only one which is connected with a user via a Maple frontend.

- **rootId** is a variable which contains the identifier of the current root node (initially *rootId* is equal to *initRootId*).

- **initNumber** is a constant which contains the number of the non-root nodes after a session initialization.

- **numberOfConnectedNodes** is a variable and it is located on shadow root nodes. This variable contains the number of those nodes which are reachable to a shadow root node during the root election.

- **shadowRoots** is a list which contains the identifier of the shadow root nodes. The order of elements in this list determines a priority order among the shadow roots, too.

- By a message **shadow:<*name, path*>** , a new shadow root node can be defined. The message has two arguments. The first argument *name* is the name (or internet address) of a machine. The second argument *path* is the access path of logging directory on this machine.

- A message **check_root** is sent to all reachable nodes by a shadow node that can contact neither to the current root nor to another shadow node which have more priority.

- A message **root_lost:<*state*>** has an argument *state* which is a boolean value. This message may be sent in two cases:

    1. If a non-root node loses the contact with the current root, it sends a `root_lost:<true>` to the first live shadow root in a list of *shadowRoots*.

    2. If a message `check_root` arrives from a shadow root k to a non-root node. In this case, if the non-root has already sent a message `root_lost:<true>` to k, nothing happens. If the non-root node can still interact with the current root or the current root although became unreachable to this node, but it can contact another shadow root with higher priority, then it sends back to shadow root k a message `root_lost:<false>` (in the implementation of the "Tolerating Root Failures" mechanism, we do not use `root_connected:` messages as in the description of its algorithm. Instead of it, the message `root_lost:`<*state*> is applied with a boolean value "false" as its argument).

- By a message **new_root**, the new root is announced in a session. Any node which receives such a message, accepts the sender of the message as the new root (see in more details below).

- In a message **scheduled_tasks:<*load, t1, t2, ...*>** , each node sends the identifiers of those tasks to the new root which are scheduled to this node for processing. Furthermore in the first argument *load*, the number of those tasks is sent which are already under processing on the node. This message is used for the initialization of load balancing mechanism (see Section 3.2.4) and a hash table *scheduled_tasks* (see Section 6.1.3) on the new root. The argument's number of this message is varying (depending on the number of scheduled tasks on a node).

- **receivedRootLost** is a list and it contains the identifiers of those nodes from which a message `root_lost:`<*state*> has arrived. A term $|receivedRootLost|$ denotes the number of elements of this list in the description of implementation.

- **trueRootLost** is a list and it contains the identifiers of those nodes from which such a message `root_lost:`<***state***> has arrived whose argument *state* is equal to value "true" A term $|trueRootLost|$ denotes the number of elements of this list in the description of implementation.

- **notReachableNodes** is a list on a shadow root node i which contains the identifiers of those non-root nodes which become unreachable to i after a root failure is detected, but before a new root is elected.

- **checkroot_sender** is a list which contains the identifier of those shadow root nodes whose sent `check_root` messages have not be answered yet.

- **existing_tasks** is a list whose elements can be task identifiers. This data structure is needed for rescheduling those tasks which were under processing:

  – either on the previous root before its failure or

  – on some other nodes which fail before the new root is announced and initialized (but the "Tolerating Non-Root Node Failures" mechanism has not be able to reschedule all the affected tasks yet, because of the root failure).

- **numberOfMessages** is an integer variable. It counts the received `scheduled_tasks:`<*load, t1, t2, ...*> messages on the new root.

- **shadowMessages** is a list which contains all `shadow:`<*name, path*> messages issued already in a session. This list is needed for the initialization of restarted nodes.

Activating and Initialization of Tolerating Root Failures

Similarly to other fault tolerant mechanisms in the system, the "Tolerating Root Failures" mechanism can be activated by the call `dist[logging](5)` after a session initialization. The effect of this command is that the Maple frontend sends a message `stable:`<5> to the root scheduler which broadcasts this message to all nodes.

The shadow root nodes in a session must be given explicitly one by one by the call `dist[shadow](name, path)`. *name* is one of the machine names which were given at the session initialization and *path* is the access path of the directory `logging` on the given machine. These given data are sent in a message `shadow:`<*name, path*> to the root scheduler which broadcasts this message to all nodes. If such a message arrives to a node, it is inserted to a list *shadowMessages* on every node. Furthermore, the identifier of the given node is determined and added to another list *shadowRoots*.

Originally, the execution of a main task of a Distributed Maple program is always attached to the Maple frontend. Therefore, source code of this task is never logged into storage system and it is unaccessible for other nodes, too. To guarantee, that a Distributed Maple program is performed until completion even if the root fails in the middle of execution, other nodes has to be able to re-run the main task. For achieving this, the source code of a main task has to be given as the argument in the call `dist[main](source)` (for an example, see Section 7.3). If such a command is issued, the Maple frontend removes the "new line" characters from the argument *source* and assigns a unique *label* to it. Then it sends the following messages to the root scheduler:

1. By sending a message `all:<cat(label, ":=proc() ", source, " end:")>`, *source* is distributed to every node as a predefined 0-ary function (the Maple function "cat" which simply concatenates its parameters is evaluated before the message is sent).

2. By sending a message `task:<t, label>`, the execution of *source* is triggered (task identifier t is generated and assigned to this task as usual). The root will schedule this task to a node.

A Distributed Maple program may contain more than one `dist[main](source)` commands, since the assigned labels are unique so are the assigned task identifiers.

Root_lost Message Handling

When a node k receives a message `root_lost:`<*state*> from another node i, k adds i to list *receivedRootLost*; and if the value of argument *state* is true, k adds i to list *trueRootLost*, too.

Check_root Message Handling

When a node i receives a message `check_root` from node j, node i adds the identifier of j to list *checkroot_sender*.

New Watchdog Modes on Non-Root Nodes

The leader election of our "Tolerating Root Failures" mechanism is implemented in two new operational modes of the watchdog mechanism located on the non-root nodes. These are called *Root_lost Mode* and *Shadow_root Mode*. If a non-root node i detects that the root became unreachable to it and reconnection time expires, its watchdog thread changes to the *Root_lost Mode*, which does the followings:

1. First of all, the watchdog thread takes from list *shadowRoots* the identifier of the first shadow root node which is not designated as dead

in hash table *crash* (see Section 6.1.3) and which can be reached by establishing a peer connection.

Since an immediate response is required for the establishing of a new peer connection (and since the recently failed peer connections are detected and closed by the watchdog mechanism before the root failure is determined), the availability of a node can be determined by checking if there is not an existing peer connection between the node and node i (if such a connection exists but fails in the meantime, node i eventually aborts, see Step 4).

2. If such a shadow root node does not exist, i aborts. If the selected identifier k is equal to i, then the watchdog thread on i changes to the Shadow_root Mode. Otherwise i sends a message `root_lost:<true>` to k.

3. It takes one by one the shadow root identifiers which is located in list *checkroot_sender* and sends a message `root_lost:<false>` to them, except to k.

4. Until a message `new_root` arrives to i, i periodically checks the connection to the chosen shadow root k similarly to the original watchdog algorithm. If k becomes unreachable to i, i aborts. If further `check_root` messages arrive to i in the mean time, they are answered also with messages `root_lost:<false>`.

   In this situation, it does not make sense to check the peer connections of i and apply the "Tolerating Peer Connection Failures" mechanism, because we cannot redirect the lost messages via the root (and neither via the selected shadow root k, since k may not become the new root). The broken peer connection is detected and the lost messages are resent only after the root election (when the watchdog thread returns to its normal operation).

5. If a message `new_root` arrives to i, its watchdog thread returns to its normal operation.

If the watchdog thread on node k selects from list *shadowRoots* a shadow root identifier which is equal to k, then it changes to the *Shadow_root Mode*. It works as follows:

1. Node k sends a message `check_root` to each node which is not designated as dead in hash table *crash* and which has not sent a message `root_lost:<true>` to k yet. Variable *numberOfConnectedNodes* is initialized with the number of dead nodes in *crash*.

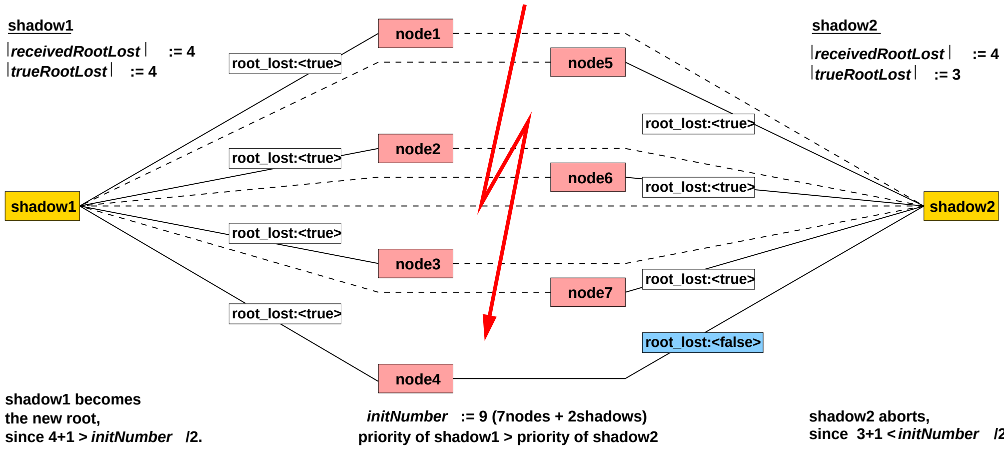

Figure 6.6: *Root election with two competitor shadow root nodes.*

2. Until the condition

$$numberOfConnectedNodes \leq |receivedRootLost|$$

or condition

$$|trueRootLost| + 1 > initNumber/2$$

becomes true, k regularly checks the connection to the reachable nodes similarly to the original watchdog algorithm. If k cannot contact to a node i anymore, k closes the peer connection to i, decrements the variable *numberOfConnectedNodes*, adds i to list *notReachableNodes*; and if i occurs in lists *receivedRootLost* or *trueRootLost*, then i is removed from these lists.

If the condition

$$numberOfConnectedNodes + 1 \leq initNumber/2$$

(the sum of the shadow root k and reachable nodes is less or equal than the half of the total number of non-root nodes in a session) becomes true in the mean time, then k aborts.

3. If the condition

$$|trueRootLost| + 1 > initNumber/2$$

(the sum of the shadow root k and the nodes which voted positively is greater than the half of the total number of non-root nodes in a session) is true , then k becomes the new root, see the case of *shadow1* on Figure 6.6. In this case, the watchdog thread of k sends a message `new_root` to the server thread of k and the watchdog thread terminates (because a watchdog thread will be started and initialized for the new root by the server thread of k). Otherwise k aborts, see the case of *shadow2* on Figure 6.6.

While the root election is performed by watchdog threads, the server threads of the nodes are not disturbed or interrupted in doing of their normal operation and the processing of already scheduled tasks is continuous on the Maple kernels.

New_root Message Handling

If a message `new_root` is received by a node k, then its handling depends on the sender of this message. If the sender is the watchdog thread of k, then k has to initialized some additional data structure to become the new root. But all other activities are common (independently such a message arrived from the local watchdog thread or from another node).

Thus, if node k receives a message `new_root` from its watchdog thread, k handles this message as the new root:

1. An empty *scheduled_tasks* hash table is created.
2. The connect thread of k is stopped.
3. The system information is loaded from the configuration file *dist.system* (which is located originally on every node).
4. A message `new_root` is broadcast to every reachable node.
5. For those node identifiers which are designated as dead in the hash table *crash*, the Insert Operation of "Restarting after Node Failures" mechanism is called (for its description, see Section 6.2.3).
6. If the list *notReachableNodes* is not empty, then the contained node identifiers are taken from *notReachableNodes* and the "Tolerating Non-Root Node Failures" mechanism is performed on them.
7. From the extensions of `descr.`*taskId* files located in the directory `logging`, all logged task identifiers are restored and loaded to a list *existing_tasks* (see Figure 6.7a).
8. A new watchdog thread is started.

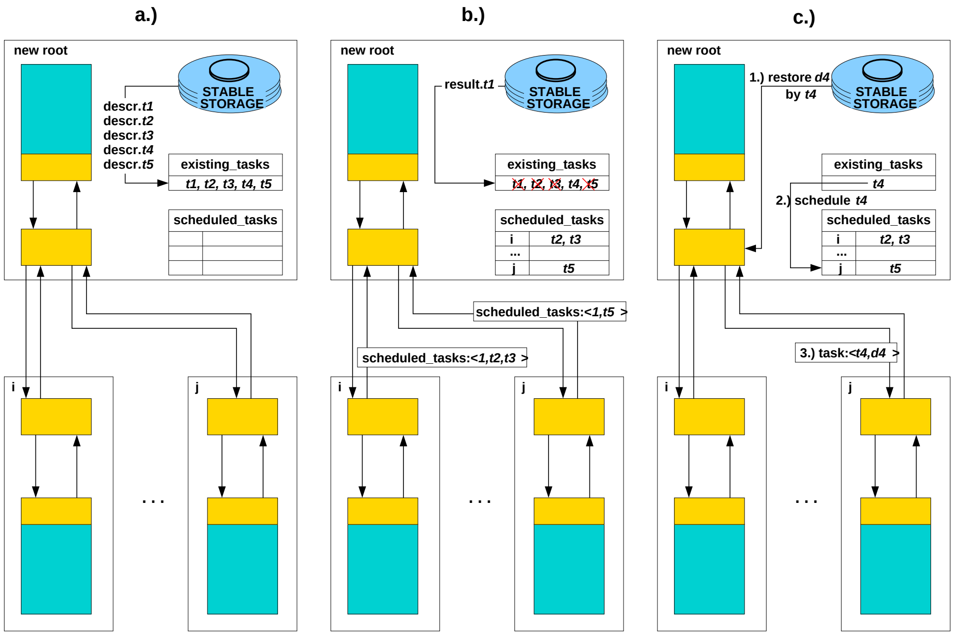

Figure 6.7: *Rescheduling after Root Failure:* ***a.)*** *Restoring all task identifiers logged in a session* ***b.)*** *Removing the identifiers of those tasks which are either already computed or still under processing.* ***c.)*** *Reschedule the remainders.*

The following steps are performed on both the new root k and all other nodes, if a message `new_root` arrives:

1. The variable *rootId* is set to k.

2. The new root k is designated as dead in hash table *crash*. Namely, *crash* refers only to the states of non-root nodes, but k is not available any more as a non-root node. Furthermore, if the new root k fails in a later time and another node becomes the root, then k will be restarted when this next new root receives a message `new_root` (see above, at Step 5 of "new_root message handling" on the new root).

3. The unanswered requests which were forwarded to the failed root are resent to the new root k in the same way as in the "Tolerating Root Failures" mechanism (see Dead Message Handling in Section 6.1.3).

4. All other kinds of messages which were sent to the failed root, but which have not been acknowledged yet are resent to the new root k in the same way as in the "Reconnecting" mechanism (see Reconnection Message Handling in Section 5.1.3)

5. By using the contents of list *running_tasks* and of hash table *externUsed*, the identifiers of all the tasks are sent in a message `scheduled_tasks:`<*load, t1, t2, ...*> to the new root k which are scheduled to the sender node for processing (see Figure 6.7b). The description of both data structures *running_tasks* and *externUsed* can be found in Section 4.3.2.

Scheduled_tasks Message Handling

When a message `scheduled_tasks:`<*load, t1, t2, ...*> arrives at the root k, the processing of this message is done as follows:

1. The variable *numberOfMessages* is incremented.
2. The received task identifier is removed from list *existing_tasks* (see Figure 6.7b).
3. The list of received task identifiers is inserted with the identifier of sender node as a key into hash table *scheduled_tasks* (see Figure 6.7b).
4. If *numberOfMessages* is equal to the number of all reachable nodes, then *numberOfMessages* is set to 0 and the identifier of all already computed tasks is restored from the extensions of `result.`*taskId* files (see Figure 6.7c). These restored task identifiers are removed from list *existing_tasks*. Afterwards, the task descriptions which are referred by the identifiers remained in *existing_tasks* are restored from the storage system and rescheduled to some nodes.

   If no task identifier remains in *existing_tasks* and hash table *scheduled_tasks* is also empty, furthermore there is no scheduled task on the root, then the session terminates.

The reason why the identifiers of computed tasks are not removed from *existing_tasks* immediately after this data structure has been initialized with the extensions of `descr.`*taskId* files (in the processing of `new_root` message) is the following. There may exist some `store:`<*t, r*> messages which are lost due to the root failure. But they are resent and processed (and their held task result are logged) before the `scheduled_tasks:`<*load, t1, t2, ...*> messages. Hereby, less tasks are rescheduled redundantly.

Modified Dead Message Handling on the Root

At the end of handling of a message `dead:`<*i*>, the root checks whether a message `scheduled_tasks:`<*load, t1, t2, ...*> has arrived from node i. If it has already arrived, then the variable *numberOfMessages* is decremented.

Otherwise the root checks whether the number of all reachable nodes became equal to *numberOfMessages* by the failure of node i. If they are equal, Step 4 of "Scheduled_Tasks Message Handling" is executed.

Modified Store Message Handling

At the end of handling of a message `store:`$<t, r>$, the session may terminate, if and only if *initRootId* is not equal to *rootId*, *existing_tasks* and *scheduled_tasks* are empty,and there is no scheduled task on the root.

In a later session, the system is able to recover the result of the whole computation by the "Logging" mechanism.

Using Together the Tolerating Root Failures and Restarting after Node Failures mechanisms

If these two mechanisms are used together, then the *Insert Operation* of the "Restarting after Node Failures" mechanism is not called immediately after a node i is designated as dead. But it is called if and only if the broadcast message `dead:`$<i>$ is acknowledged (see Section 5.1) by at least the half of the total number of non-root nodes.

If a node is restarted, every message `shadow:`$<name, path>$ which is stored in the list *shadowMessages* is sent in order to the restarted node.

Chapter 7

Examples and Test Experience

In the previous chapters, we described the theoretical background and functionality of the fault tolerance mechanisms in Distributed Maple. Now, we present how they can be used.

First, we give a short overview about new commands, which are needed for using the new features. Then, we present by examples the usage of them. At the end of the chapter, we summarize our test experience related to fault tolerance in Distributed Maple.

7.1 New Commands

Since the new fault tolerance features of Distributed Maple are transparent to Maple programs (the programs need not be changed for using them). The only exception is the "Tolerating Root Failures" mechanism (see below).

All fault tolerant mechanisms integrated in Distributed Maple can be switched on and off by the command

`dist[logging]`(*level*);

where the parameter *level* sets the fault tolerant level of the system. Its value can be 0, 1, 2, 3, 4 or 5 where 0 switches off all fault tolerant mechanisms; the values from 1 to 5 activate more and more fault tolerant mechanisms. Namely, if a higher fault tolerant level is chosen, then all fault tolerant mechanisms on the lower levels are automatically switched on, too (because the mechanisms on the higher levels use the features of the mechanisms on the lower levels):

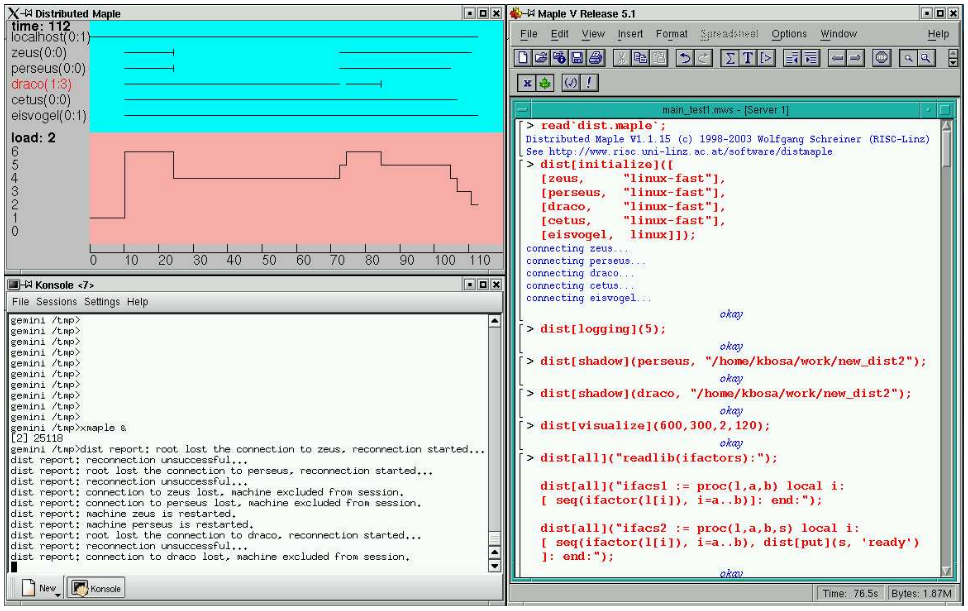

Figure 7.1: The user is informed about fault tolerance activities

- 1 . . . switches on the "Fast Mode" of the "Logging" mechanism,
- 2 . . . switches on the "Safe Mode" of the "Logging" mechanism,
- 3 . . . switches on the "Tolerating Non-Root Node Failures" and "Tolerating Peer Connection Failures" mechanisms,
- 4 . . . switches on the "Reconnecting" and "Restarting after Node Failures" mechanisms,
- 5 . . . switches on the "Tolerating Root Failures" mechanisms.

This command may be issued only after the *dist.maple* interface to the distributed backend is loaded from the file `dist.maple` and the distributed session is established by the command `dist[initialize]`.

Usage of "Logging" When the user activates one of logging modes, the system starts to save the results of concurrent tasks to a directory `logging`. If the system fails, the user has to restart it. A second (or a later) execution restores the already computed and logged task result. When the computation finishes and `dist[terminate]` command is executed, the directory `logging` is removed.

The logging and recovery mechanism is completely transparent to Distributed Maple sources. If a task is given (a) new identifier(s) in a second, third or later execution, the task can be referred by each of them (see Section 7.2).

We suggest to use always the Safe Mode which supports higher-order tasks and whose extra overhead compared with the Fast Mode is negligible.

Usage of "Tolerating Non-Root Node Failures" and "Tolerating Peer Connection Failures" These mechanisms monitor the connections between the nodes in the background and reschedule the corresponding tasks or resend the corresponding messages if necessary. The user is informed about their activities by messages. In that case, if on-line visualization is switched on and a machine becomes unreachable, its name turns red in the visualization window (see Figure 7.1).

The "Tolerating Non-Root Node Failures" mechanism cannot make a distinction between connection failure and node failure. If a connection fails between the root and a node, the corresponding tasks will be redirected and the node will be lost to the computational session as in the case of node failures. It may also happen, that a non-root node detects earlier than the root the failure of another node and the message "lost peer connection" is displayed before the user is informed about the node failure.

If we assume that only Stop failures may occur, the system is able to run on this fault tolerance level till the root node crashes or the session is terminated by the user. However, if too many nodes become unreachable to the root, it does not make sense to let the session run further. In such a situation, the user should kill the processes of the root (if the root is unreachable, the remaining nodes shut themselves down) and restart the session (from a saved state).

Usage of "Reconnecting" and "Restarting after Node Failure" As we have mentioned earlier, the main purpose of these mechanisms are reducing the loss of resources after failures. By using the "Reconnecting" mechanism, the execution time of the computation may be reduced. Namely, the tasks which are already under processing (or perhaps finished) on a disconnected node do not need to be rescheduled and computed afresh.

If the reconnection is unsuccessful and the unreachable node is excluded from the session, the root attempts to restart it after some time. If it does not succeed, it waits some time and tries again, and so on. If the reconnection or the restarting of a node is successful, the user is informed about it by messages and name of the machine turns back to black in the visualization window (see Figure 7.1).

Usage of "Tolerating Root Failures" Usage of this mechanism is a bit more complicated than the others mentioned above i.e. it is not enough to switch it on. On the one hand, the user should give the name of those machines (called shadow roots) that may be able to substitute for the original root in case of need. These machines have to be able to access to the directory `logging`. A machine can be declared as a shadow root with the command

$$\texttt{dist}[\texttt{shadow}](\textit{machineName}, \textit{pathOfLogDir});$$

where *machineName* is the name of the machine which is the same as was given to the command `dist[initialize]`; and *pathOfLogDir* is the access path of the directory `logging` on the machine. This command can be issued either before or after command `dist[logging]`, too (but only after the command `dist[initialize]`).

On the other hand, if the user wants to be sure that her or his Distributed Maple program will be performed until completion (even if, the root fails in the middle of execution), all of the non-root nodes have to be able to re-run the source code of main task. By the command

$$\texttt{dist}[\texttt{main}](\textit{source});$$

the Maple commands denoted by the parameter *source* is propagated to all nodes; the execution of these commands is started immediately on one of them.

So, if the original root becomes unreachable to a distributed session, the remaining nodes are able to undertake its role. Within a session, the root node may change more than once. Since the usage of this mechanism assumes the working of all other fault tolerance mechanisms mentioned previously, all different kinds of stop failures are also handled on this fault tolerance level of the system. However, it still can occur that the system fails, if too many node fails at the same time. In this case, the user must restart the session and continue the computation from a saved state restored by "Logging" mechanism.

7.2 A Simple Example for Logging

In this very simple example, we present the working of the logging mechanism. As usual, we load the file `dist.maple` and initialize the distributed session first. Our current session consists of the local host and the machine `zeus`. Then we switch on the Safe Mode of "Logging" and start to execute a task (Maple returns immediately an assigned identifier for the started task).

But before we would query the result of the task, we trigger a session failure off.

```
gemini!5>maple
    |\^/|     Maple V Release 5.1 (Universitaet Linz)
._|\|   |/|_. Copyright (c) 1981-1998 by Waterloo Maple Inc. All rights
 \  MAPLE  /  reserved. Maple and Maple V are registered trademarks of
 <____ ____>  Waterloo Maple Inc.
      |       Type ? for help.
> read`dist.maple`;
Distributed Maple V1.1.15 (c) 1998-2001 Wolfgang Schreiner (RISC-Linz)
See http://www.risc.uni-linz.ac.at/software/distmaple
> dist[initialize]([[zeus,linux]]);
connecting condor...
                                     okay

> dist[logging](2);
                                     okay
> t1 := dist[start](int, x^n, x);
                                     t1 := 0
> # ---------------------- SESSION FAILURE ------------------
```

If we assume that the computation of the task was finished before the session failure, the content of directory `logging` may consist of the following files: `descr.0, result.0, sessionid` and `taskid.1332700025`. The file `descr.0` and file `result.0` contains the description and the computed result of the task (as we can see the extension of both files is the task identifier). For the roles of files `sessionid` and file `taskid.1332700025`, see Section 4.2.2 and Section 4.3.2. For accessing the result of the task, we have to restart the system.

```
gemini!5>maple
    |\^/|     Maple V Release 5.1 (Universitaet Linz)
._|\|   |/|_. Copyright (c) 1981-1998 by Waterloo Maple Inc. All rights
 \  MAPLE  /  reserved. Maple and Maple V are registered trademarks of
 <____ ____>  Waterloo Maple Inc.
      |       Type ? for help.
> read`dist.maple`;
Distributed Maple V1.1.15 (c) 1998-2001 Wolfgang Schreiner (RISC-Linz)
See http://www.risc.uni-linz.ac.at/software/distmaple
> dist[initialize]([[zeus,linux]]);
connecting condor...
                                     okay

> dist[logging](2);
                                     okay
```

```
> t1 := dist[start](int, x^n, x);
                                t1 := 134217728

> r := 1 + dist[wait](0);
                                    (n + 1)
                                   x
                          r := 1 + --------
                                    n + 1

> r := 1 + dist[wait](134217728);
                                    (n + 1)
                                   x
                          r := 1 + --------
                                    n + 1

> dist[terminate];
                                okay

> quit;
```

In case of restarting, we issue all the commands from loading `dist.maple` to starting the task. The system assigns a new identifier for the task, but it does not compute it again. If we check again the content of directory `logging`, we find an additional file there, called `link.134217728`. The content of this file is the original task identifier and its extension is the new task identifier. By using this file, the task result computed in a previous session can be restored by the new identifier, too. This feature may become crucial, if a higher-order task (see Section 4.3) refers an already computed task after the session has been restarted at least once.

So if we want to display the result of the task, we must issue a command `dist[wait]` with one of the task identifiers as a parameter. The system checks whether a result file or a link file with the given identifier as an extension exists in the directory `logging` and recovers the result from them.

At the end, by issuing command `dist[terminate]`, the session is closed and the directory `logging` is deleted.

7.3 Usage of Tolerating Root Failures

Our next example is a little bit more complicated. We generate 100 of numbers whose order of magnitude are around 10^{32} and at the end this program returns the complete integer factorization of them. Such a computation without failures usually takes approximately one and half minutes on a distributed session that consists of 5 machines (local host + 4 remote nodes) where the machines are Intel Celeron 733 Mhz with 256 MB RAM.

```
gemini!5>maple
    |\^/|     Maple V Release 5.1 (Universitaet Linz)
._|\|   |/|_. Copyright (c) 1981-1998 by Waterloo Maple Inc. All rights
 \  MAPLE  /  reserved. Maple and Maple V are registered trademarks of
 <____ ____>  Waterloo Maple Inc.
      |       Type ? for help.
> read`dist.maple`;
Distributed Maple V1.1.15 (c) 1998-2001 Wolfgang Schreiner (RISC-Linz)
See http://www.risc.uni-linz.ac.at/software/distmaple
> dist[initialize]([
> [perseus, linux],
> [aquila,  linux],
> [draco  , linux],
> [cetus  , linux]
> ]);
connecting perseus...
connecting zeus...
connecting draco...
connecting cetus...
                                   okay

> dist[logging](5);
                                   okay

> dist[shadow](perseus, "/home/kbosa/work/new_distmaple");
                                   okay

> dist[shadow](draco, "/home/kbosa/work/new_distmaple");
                                   okay

> dist[all]("readlib(ifactors):");
                                   okay

> dist[all]("ifacs := proc(l,a,b) local i; [ seq(ifactor(l[i]), i=a..b) ]: end:");
                                   okay

> t:=dist[main]("local random, l, t1, t2, t3, t4, r1, r2, r3, r4;
>
> readlib(randomize); randomize(4711);
> random := rand(10^32);
> l := [ seq(random(), i=1..100) ];
>
> t1 := dist[start](ifacs, l,  1, 25);
> t2 := dist[start](ifacs, l, 26, 50);
> t3 := dist[start](ifacs, l, 51, 75);
> t4 := dist[start](ifacs, l, 76,100);
>
> r1 := dist[wait](t1);
> r2 := dist[wait](t2);
```

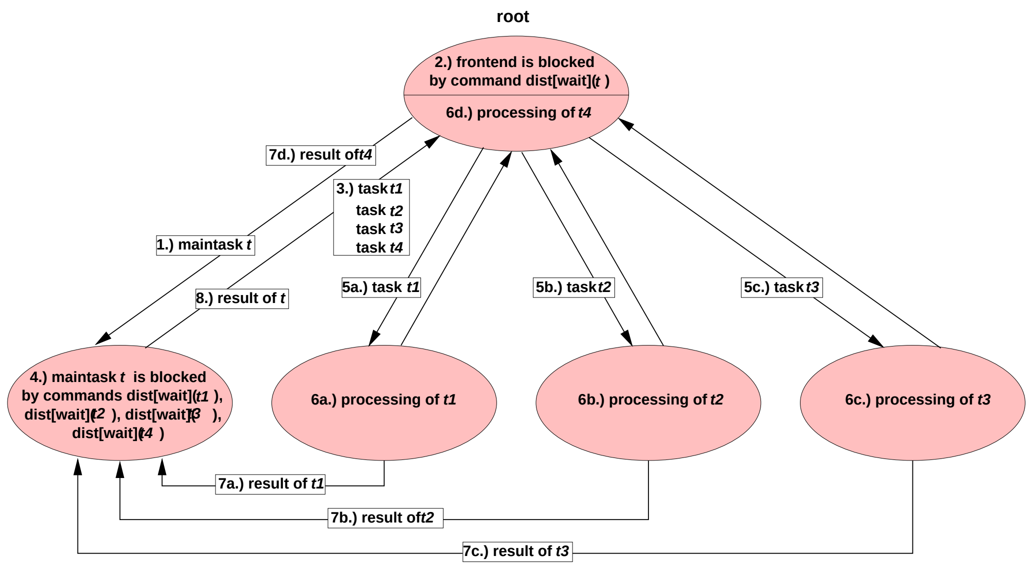

Figure 7.2: Execution Model for the Example in Section 7.3

```
> r3 := dist[wait](t3);
> r4 := dist[wait](t4);
>
> [ op(1..25, r1), op(1..25, r2), op(1..25, r3), op(1..25, r4) ];
> ");
                                     t := 0
> w := dist[wait](t);

# ... Here comes the result of factorization of the hundred numbers...

> dist[terminate]();
                                     okay
```

First of all, we establish the session with 5 nodes, then we activate all the fault tolerance mechanisms in Distributed Maple (fault tolerance level 5). Afterwards, we enumerate the shadow root nodes and we give the local access path of directory `logging` on each such nodes.

With the command `dist[all]`, we load the library *ifactors* of Maple to each node and by using also this command we define a new function, called *ifacs* on every node which simple calls the Maple function *ifactor* for some elements of a list one by one.

The main function is defined as the parameter of command `dist[main]`. Its structure is like an ordinary function in Maple (e.g.: it may contain local variables), except on thing: it cannot be finish with a command "*end;*" (since it is given between brackets), but it must return a value. Our current main function generates the 100 numbers first and then distributes them among 4 concurrent tasks equally. Each task is scheduled to some node where it runs the function *ifacs* to compute the factorization of the given numbers. The execution model of this program (without any failure) is depicted in Figure 7.2; this model assumes that the default configuration of Distributed Maple is used: maxload = 0 and minload = 0 (see Section 3.2.4).

In case of root failure, one of the shadow root becomes the new root and the session finishes the whole computation. In such a situation, we can display the result if we restart the session and issue the request `dist[wait]` which refers to the main task.

7.4 Test Experience

The effect of fault tolerance mechanisms on the performance of Distributed Maple has not been measured yet, but we do not expect it to be significant. For instance in the case of the "Logging" 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" 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 in case of the other mechanisms is also not significant, because these mechanisms maintain only some small extra data structures both on root node and on each non-root node. They do not use additional messages during the normal operation.

In the testing phase, we executed some long-running computations, whose solutions are required approximately 3 or 4 days. During these executions, we triggered several *Stop* failure situations (both root and non-root failures) and the system was able to tolerate these kinds of failures and finish the computations.

On the average only one long-running test from every 10 failed with corrupted message (message failure). But in such cases, the computations could always be continued from a saved state of the session.

Chapter 8

Conclusions

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. With these developments, Distributed Maple is by far the most advanced system for computer algebra concerning reliability in distributed environments. The improvement of the system is demonstrated by Figure 8.1.

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.

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,

Stop failures		without fault tolerance	using fault tolerance
the root node crashes		system aborts	system continues the normal operation
a non−root node crashes	less than half or half of non−root nodes crash within a certain time bound	deadlock	system continues the normal operation
	more than half of non−root nodes crash within a certain time bound		system aborts
	the root and all shadow root nodes crash within a certain time bound		system aborts
one or more connections between the root and non−root nodes break		deadlock	system continues the normal operation
one or more connections between non−root nodes break		deadlock	system continues the normal operation
one or more Maple processes fail		deadlock	deadlock

Figure 8.1: Assessment of Distributed Maple Fault Tolerance

it is suitable for the system environments that we have used up to now (≤ 30 nodes); many parallel computer algebra applications do not scale well enough to profit from considerably more nodes.

8.1 Comparison with Checkpointing

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 [37] (see Figure 8.2). P-GRADE is a parallel programming environment which supports the whole life-cycle of parallel program development [51]. 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 2.2.2, such an approach is relatively complex

	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)

Figure 8.2: Comparison of Fault Tolerance in Distributed Maple and P-GRADE Checkpointing

because it is targeted to general parallel applications. In the case of our "Logging" 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, it cannot use any already saved information of this checkpoint and it can be restarted only from a previous checkpoint. The "Logging" 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 normal operation without human intervention.

Furthermore, an additional advantages of applying of functional task model is that the normal operation on faultless nodes is always continuous, because it does not have to be interrupted or suspended due to handling and tolerating of any failure situations detected in a session. Both systems are centralized, therefore the scalability of these systems is an open question.

8.2 Plans for Future Developments

Distributed Maple shows that the application of cluster computing to computer algebra is able to extend the range of solvable problems. However, its centralized architecture is not scalable beyond a certain number of nodes. This fact is the next difficulty to overcome before we can enlarge further the range of those problems to which the parallel computer algebra algorithms implemented in this system are applicable.

To achieve more scalability, it would be a good idea to make the system use an underlying grid infrastructure [23], e.g. the Globus [24] middleware (similarly to the case of PVMaple [49, 50]). By using grid services, numerous computers may become available for Distributed Maple sessions.

Of course the chance to access a large number of resources by invoking grid services is not enough in itself to that the system can become (largely) scalable. The architecture and the communication protocol of Distributed Maple also have to be modified and rearrange for handling such a large number of resources. Furthermore, it is very important to keep and adopt our fault tolerant achievements to such a proposed scalable architecture.

A possible answer can be the applying of a *hierarchical programming model* by which the original execution model of Distributed Maple can be extended to a tree of computational sessions and subsessions. The core idea is that when a frontend executes a Distributed Maple algorithm, it requests from some kind of grid middleware a set of resources which subsequently form the corresponding computational session. If in the course of the algorithm a parallel subalgorithm is invoked on any computational node of this session, this node may allocate new resources from the grid middleware and build a subsession for the execution of the subalgorithm. In this hierarchical programming model, a tree of sessions can be dynamically constructed whose resources are collected from all over the Internet.

Since the structures of such a computational session and its subsessions separately are very similar to the original architecture of the Distributed Maple, our fault tolerance mechanisms presented in the previous chapters can be extended to such a proposed environment by minor modification; this could demonstrate how a distributed system based on functional tasks simplifies some problems in grid computing.

Bibliography

[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] L. Alvisi, T. Bressoud, A. El-Khashab, K. Marzullo, and Z. Zagorodnov. *Wrapping server-side tcp to mask connection failures.* In Proceedings of Infocom 2001, April 2001.

[3] 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.

[4] 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.

[5] 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.

[6] Laurent Bernardin. *Maple on a Massively Parallel, Distributed Memory Machine.* In M. Hitz and E. Kaltofen, editor, PASCO'97 — Second International Symposium on Parallel Symbolic Computation, pages 217–222, Maui, Hawaii, July 20-22, 1997. ACM Press, New York.

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

[8] Károly Bósa, Wolfgang Schreiner. *Task Logging, Rescheduling, and Peer Checking in Distributed Maple.*Technical Report 02-10, Research Institute For Symbolic Computation (RISC-Linz), Johannes Kepler University, Linz, Austria, March 2002.

[9] Károly Bósa, Wolfgang Schreiner. *Tolerating Stop Failures in Distributed Maple*. In Distributed and Parallel Systems – Cluster and Grid Computing, DAPSYS'2002, 4th Austrian Hungarian Workshop on Distributed and Parallel Systems, pages 203–210, Linz, Austria, September 29 – October 02, 2002. Kluwer Academic Publishers,Boston.

[10] Károly Bósa, Wolfgang Schreiner. *Tolerating Stop Failures in Distributed Maple*. Parallel and Distributed Computing Practices, special issue on Dapsys 2002, 15 pages, NovaPublishers, 2003 (to appear).

[11] N. Budhiraja and K. Marzullo. *Highly-available services using the primacy-backup approach*. In Proceedings of the 2nd Workshop on Management of Replicated Data, Monterey, CA, 1992.

[12] 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.

[13] K. C. Chan, A. Diaz,, and E. Kaltofen. *A Distributed Approach to Problem Solving in Maple*. In R. J. Lopez, editor, Maple V: Mathematics and its Application, Proceedings of the Maple Summer Workshop and Symposium (MSWS'94), pages 13–21, Boston, 1994. Birkhäuser.

[14] T. Chandra and S. Toueg. *Unrealiable Failure Detectors for Reliable Distributed Systems*. Journal of the ACM, 43(2): 225-267, 1996.

[15] Bruce Char. *Progress Report on a System for General-Purpos Parallel Symbolic and Algebraic Computation*. In ISSAC'90 Int. Symp. on Symbolic and Algebraic Computation, pages 96–103, Tokyo, Japan, August 20-24, 1990. ACM Press.

[16] Bruce Char and Jeremy Johnson. *Some Experiments with Parallel BigNum Arithmetic*. In M. Hitz and E. Kaltofen, editor, PASCO'97 — Second International Symposium on Parallel Symbolic Computation, pages 94–103, Maui, Hawaii, July 20-22, 1997. ACM Press, New York.

[17] 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.

[18] A. Diaz, E. Kaltofen, K. Schmitz, and T. Valente. *DSC — A System for Distributed Computation.* In M. WWatt, editor, Proceedings of ISSAC'91, pages 324–333. ACM Press, 1991.

[19] Angel Diaz and Erich Kaltofen. *FoxBox: A System for Manipulating Symbolic Objects in Black Box Representation.* In O. Gloor, editor, ISSAC'98 International Symposium on Symbolic and Algebraic Computation. ACM Press, New York, 1998.

[20] J. Della Dora and J. Fitch, editors. *Computer Algebra and Parallism.* Academic Press, London, UK, 1989.

[21] 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.

[22] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. *Impossibility of distributed consenssus with one faulty process.* Journal of the ACM, 32(2):374-382, April 1985.

[23] I. Foster, C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure.* Morgan Kaufmann Publishers, 1999.

[24] I. Foster, C. Kesselman. *The Globus Toolkit. In The Grid: Blueprint for a New Computing Infrastructure.* Morgan Kaufmann Publishers, 1999. pp. 259–278.

[25] Hector Garcia-Molina. *Elections in a distributed computing system.* IEEE Transactions on Computers, C-31(1):47–59, January 1982.

[26] Thierry Gautier, Hoon Hong, Jean-Louis Roch, and Wolfgang Schreiner. *Parallel Implementation.* In V. Weispfennig, J. Grabmeier, E. Kaltofen, editor, Handbook of Computer Algebra — Foundations, Applications, Systems, chapter 2.16. Springer, Heidelberg, 2002.

[27] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderam. *PVM: Parallel Virtual Machine — A User's Guide and Tutorial for Network Parallel Computing.* MIT Press, Cambridge, MA, 1994.

[28] 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.

[29] Markus Hitz and Erich Kaltofen, editors. PASCO'97 — Second International Symposium on Parallel Symbolic Computation, Maui, Hawaii, July 20-22, 1997. ACM Press, New York.

[30] Hoon Hong, editor. PASCO'94 — First International Symposium on Parallel Symbolic Computation, Hagenberg/Linz, Austria, September 26-28, 1994. World Scientific Publishing, Singapore.

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

[32] 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.

[33] 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.

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

[35] M.F. Kaashoek and A.S. Tanenbaum. *Group communication in the Amoeba distributed operating system*. In Proc. IEEE 11th International Conference on Distributed Computing Systems (ICDCS), pages 222–230. IEEE Computer Society Press, 1991.

[36] James Arthur Kohl and Philip M. Papadopoulos. *Efficient and Flexible Fault Tolerance and Migration of Scientific Simulations Using CUMULVS*. Symposium on Parallel and Distributed Tools, Welches, Oregon, United States. Pages: 60–71, 1998.

[37] 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.

[38] V. Kumar, A. Y. Grama, V. N. Rao. *Scalable Load Balancing Techniques for Parallel Computers.* Journal of Parallel and Distributed Computing, 1994.

[39] 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.

[40] 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.

[41] Nancy A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, Inc. San Francisco, California, 1996.

[42] Manish Marwah, Shivakant Mishra, and Christof Fetzer. *TCP Server Fault Tolerance Using Connection Migration to a Backup Server.* Appeared in Proc. of IEEE Int. Conf. on Dependable Systems and Networks (DSN 2003), San Francisco, CA, June 22–25, 2003.

[43] T. Metzner, M. Radimersky, A. Sorgatz, and S. Wehmeier. *User's Guide to Macro Parallelism in MuPAD 1.4.1.*. Teubner. Stuttgart, Germany, 1999.

[44] 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.

[45] Michael Munk and Franz Winkler. *CASA — A System for Computer Aided Constructive Algebraic Geometry.* In J. Calmet and C. Limongelli, editors, DISCO'96 — International Symposium on the Design and Implementation of Symbolic Computation Systems, volume 1128 of LNCS, pages 297–307, Karsruhe, Germany, 1996. Springer, Berlin.

[46] Taesoon Park and Heon Y. Yeom. *Application Controlled Checkpointing Coordination for Fault-Tolerant Distributed Computing Systems.* Parallel Computing, 24(4):467–482, March 2000.

[47] Cleopatra Pau and Wolfgang Schreiner. *Distributed Mathematica — User and Reference Manual*. Technical Report 00-25, RISC-Linz, Johannes Kepler University, Linz, Austria, July 2000.

[48] Dana Petcu. *PVMaple, a Distributed Approach to Cooperative Work of Maple Processes*. In Recent Advances in Parallel Virtual Machine and Message Passing Interface, 7th European PVM/MPI User's Group Meeting, volume 1908 of Lecture Notes in Computer Science, pages 216–224, Balatonfüred, Lake Balaton, Hungary, September 10-13, 2000.

[49] D. Petcu, D. Dubu, M. Paprzycki. *Towards a Grid-Aware Computer Algebra System*. LNCS 3038, eds. M. Bubak, J. Dongarra, Springer (2004), 499–502.

[50] D. Petcu, D. Dubu, M. Paprzycki. *Extending Maple to the Grid*. Design and Implementation. Submitted to ISPDC'2004, Cork, Ireland, July 5–7, 2004.

[51] *P-GRADE environment*: http://www.lpds.sztaki.hu/projects/pgrade.

[52] 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.

[53] B. Randell, P. Lee, and P. Treleaven. *Reliability issues in computing system design*. ACM Computing Surveys, 10(2):123–166, Jun 1978.

[54] Jean Louis Roch and Gilles Villard. *Parallel Computer Algebra*. Lecture Notes for a Tutorial, ISSAC'97, Hawaii, July 1997. http://www.apache.imag.fr/~jlroch/ps/97-issac.ps.gz.

[55] 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.

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

[57] Wolfgang Schreiner. *A Para-Functional Programming Interface for a Parallel Computer Algebra Package*. Journal of Symbolic Computation, 21(4–6):593–614, 1996.

[58] Wolfgang Schreiner. *Distributed Maple — User and Reference Manual.* Technical Report 98-05, RISC-Linz, Johannes Kepler University, Linz, May 1998. http://www.risc.uni-linz.ac.at/software/distmaple.

[59] Wolfgang Schreiner. *Developing a Distributed System for Algebraic Geometry.* In Barry H.V. Topping, editor, EURO-CM-PAR'99 Third Euroconference on Parallel and Distributed Computing for Computational Mechanics, pages 137–146, Weimar, Germany, March 20-25, 1999. Civil-Comp Press, Edinburgh.

[60] 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.

[61] Wolfgang Schreiner, Cristian Mittermaier, and Franz Winkler. *Analyzing Algebraic Curves by Cluster Computing.* In Peter Kacsuk and Gabriele Kotsis, editors, DAPSYS'2000, 3rd Austrian-Hungarian Workshop on Distributed and Parallel Systems, pages 179–188, Balatonfüred, Lake Balaton, Hungary September 10-13, 2000. Kluwer Academic Publishers.

[62] Wolfgang Schreiner, Cristian Mittermaier, and Franz Winkler. *On Solving a Problem in Algebraic Geometry by Cluster Computing.* In Arndt Bode, Thomas Ludwig, and Roland Wismüller, editors, Euro-Par 2000, European Conference on Parallel Computing, Lecture Notes in Computer Science, Munich, Germany, August 29-September 1, 2000. Springer, Berlin.

[63] Wolfgang Schreiner. *Manager-Worker Parallelism versus Dataflow in a Distributed Computer Algebra System.* PaCT'2001, Parallel Computing Technologies, Sixth International Conference, September 3-7, 2001, Novosibirsk, Russia. Lecture Notes in Computer Science 2127, pp. 329–343, Springer, Berlin.

[64] 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.

[65] Wolfgang Schreiner, Károly Bósa, and Gábor Kusper. *Introducing Fault Tolerance to Distributed Maple.* Technical Report 01-03, Research Institute For Symbolic Computation (RISC-Linz), Johannes Kepler University, Linz, Austria, January 2001.

[66] 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.

[67] Kurt Siegl. *Parallelizing Algorithms for Symbolic Computation Using ||MAPLE||*. In Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pages 179–186, San Diego, California, May 19-22, 1993. ACM Press.

[68] 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.

[69] Scott D. Stoller. *Leader Election in Asynchronous Distributed Systems.* Appeared in IEEE Transactions on Computers, Vol. 49No. 3; March 2000, pp. 283–284.

[70] S. Sultan, K. Srinivasan, D. Iyer, and L. Iftode. *Migratory TCP: Connection migration for service continuity over the internet.* In Proceedings of the 22th IEEE International Conference on Distributed Computing Systems, Vienna, Austria, July 2002.

[71] Dongming Wang. *On the Parallelization of Characteristic-Set-Based Algorithms.* In H. P. Zima, editor, Parallel Computation — First International ACPC Conference, volume 591 of Lecture Notes in Computer Science, pages 338–349. Springer, Berlin, 1991.

[72] Waterloo Maple. Maple 6, 2001. http://www.maplesoft.com

[73] R. E. Zippel. *Computer Algebra and Parallism.* Lecture Notes in Computer Science, Ithaca, USA, May 1990. Springer-Verlag.

[74] H. Zou and F. Jahanian. *Real-time primary-backup (RTPB) replication with temporal consistency guarantees.* In Proceedings of the 18th International Conference on Distributed Computing Systems, 1998.

List of Figures

Index

CURRICULUM VITAE

KÁROLY BÓSA

Research Institute for Symbolic Computation (RISC)
Johaness Kepler University, Linz, Austria

Tel: +43 (0)732 2468 9984 Fax: +43 (0)732 2468 9930
E-mail: `kbosa@risc.uni-linz.ac.at`
Homepage: `http://www.risc.uni-linz.ac.at/people/kbosa`

Personal Data

Family name:	Bósa
First name:	Károly
Date of birth:	22nd December, 1975
Place of birth:	Miskolc, Hungary
Citizenship:	Hungarian
Private address:	A-4232 Hauptstrasse 62., Hagenberg, Austria

Education

1999 – :	Currently, I am a Ph.D. Student of Technical Sciences. Research Institute for Symbolic Computation (RISC-Linz), Johannes Kepler University, Linz, Austria. Research topic: Parallel Computing and Fault Tolerance Mechanisms.
1994 – 1999:	M.Sc. in Computer Science. University of Arts and Sciences "Kossuth Lajos" (KLTE), Debrecen, Hungary. Diploma Thesis: The MNEWS Information System.
1990 – 1994:	High School "Földes Ferenc", Miskolc, Hungary I passed the final exam with excellent marks.

Research Interests

- Parallel and distributed computing
- GRID computing
- Parallel functional programming
- Fault Tolerance
- Formal methods in computer science

Journal Publication

Károly Bósa, Wolfgang Schreiner.
Tolerating Stop Failures in Distributed Maple.
Parallel and Distributed Computing Practices, special issue on DAPSYS 2002, 15 pages, NovaPublishers, 2003 (to appear).

Wolfgang Schreiner, Christian Mittermaier, Károly Bósa.
Distributed Maple: Parallel Computer Algebra in Networked Environments.
Journal of Symbolic Computation, volume 45, number 3, pp. 305-347, Academic Press., 2003.

Refereed Publications

Károly Bósa, Wolfgang Schreiner.
Tolerating Stop Failures in Distributed Maple.
DAPSYS 2002, 4th Austrian-Hungarian Workshop on Distributed and Parallel Processing, Linz, Austria, September 29-October 2, 2002, Kluwer Academic Publishers, Boston, 8 pages.

Wolfgang Schreiner, Károly Bósa, Gábor Kusper.
Fault Tolerance for Cluster Computing Based on Functional Tasks.
Euro-Par 2001, European Conference on Parallel Computing, Manchester, UK, August 28 - August 31, 2001. Lecture Notes in Computer Science, Springer, Berlin, 5 pages, Springer-Verlag.

Technical Reports

Károly Bósa, Wolfgang Schreiner.
Task Logging, Rescheduling, and Peer Checking in Distributed Maple.
Technical Report, RISC-Linz, Johannes Kepler University, Linz, Austria, March 2002.

Wolfgang Schreiner, Károly Bósa, Gábor Kusper.
Introducing Fault Tolerance to Distributed Maple.
Technical Report 01-03, RISC-Linz, Johannes Kepler University, Linz, Austria, January 2001.

Professional Specifications

- I have 6 years experience in Object Oriented Technology and Java programming.
- For my Ph.D. thesis, I developed Java software. (Fault Tolerance for Distributed Maple).
- For seven months, I have participated in the development of the Panasonic mobile phone X60 at the COMNEON GmbH. In this project, I worked with C++.
- For my diploma work, I also developed Java software. (The MNEWS Information System).

More:

Operating systems:	DOS, Windows (98/NT/2000), UNIX(Linux)
Programming languages:	Pascal, C/C++, Java, and Assembler
Database handlers:	Oracle(SQL), MS Access
Others:	HTML, XML, TEX, and Perl

Other Professional Experience

September 2003 - March 2004: Working at COMNEON GmbH. and participating in the development of Panasonic mobile phone X60. *Topic*: Member of a group whose task was the integration of Teleca's WAP/MMS application into the Cell Phone Operating System APOXI.

November 2001 - May 2002: Member of the System Administration Group of RISC-Linz.
Topic: Installation and maintains of Windows and Linux systems. Configuration and testing new softwares (VMware, Xfree86-4, port scanners, etc.).

March 2001-June 2001: Scholarship of Foundation Action Austrian-Hungarian.

Since October 1999, I am a Ph.D. student at RISC-Linz in Austria.

September 1998-December 1998: OMFB project and TDK (Scientific Student Forum) work.
Topic: Development of network applications in Java (client-server systems managing databases). Special Award, Hungarian Scientific Student Forum (OTDK) 1999.

July 1998: Summer practice at Mol Rt., Budapest.

Topic: Network organization, network maintenance, network administration: Windows NT, ORACLE.

October 1997-February 1998: University-GH Paderborn with Tempus scholarship (Germany).
Topic: Problems of modulo arithmetics.

May 1997-July 1997: University-GH Paderborn with Tempus scholarship (Germany).
Topic: Problems of modulo arithmetics and Java programming.

1996-1999: Teaching at the Institute of Mathematics and Informatics, KLTE (University of Arts and Sciences "Kossuth Lajos").
Topic: General Informatics, Programming languages, Database handling.

February 1996: Participation on the Hungarian Programming Championship, Miskolc, Hungary.

Languages

Hungarian:	native
English:	good
German:	beginner