TOLERATING STOP FAILURES IN DISTRIBUTED MAPLE*

Károly Bósa, Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC-Linz)
Johannes Kepler University, Linz, Austria
http://www.risc.uni-linz.ac.at
`FirstName.LastName@risc.uni-linz.ac.at`

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

Keywords: distributed systems, fault tolerance, computer algebra.

1. Introduction

Distributed Maple is a system for writing parallel programs in the computer algebra system Maple. We have successfully employed this system for the parallelization of various methods and applications in algebraic geometry [5]. The system is portable and has been used in various kinds of parallel and distributed computing environments.

Numerous faults may occur during the execution of Distributed Maple: *Message* failures, *Stop* failures or *Byzantine* failures [3]. We focus on *Stop* failures and we assume in this paper that only such failures may occur. In more detail, we deal with the following error situations: the root crashes, some non-root node crashes, some Maple process fails, a

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

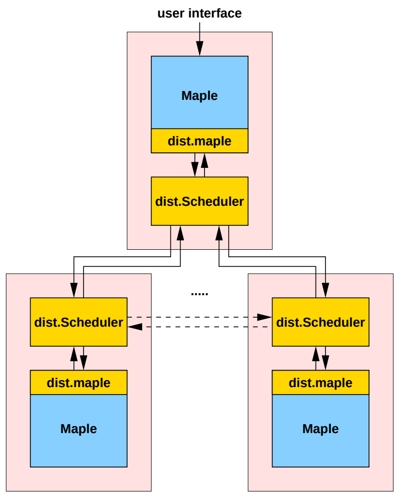

Figure 1. System Model

connection between the root and another node breaks and a connection between two non-root nodes breaks.

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. We say that a node crashes if the machine stops execution or reboots, or if the Distributed Maple process aborts.

Earlier, we have introduced three fault tolerance mechanisms, by which some of the mentioned *Stop* failure problems are covered. The functional task model of Distributed Maple has considerably simplified these developments [6].

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

With these mechanisms, Distributed Maple is by far the most advanced system for computer algebra concerning reliability in distributed environments. Our goal is to extend the time scale of distributed computations (measured in house at the moment) to many days as is required for realistic applications.

2. System and Execution Model

A session comprises a set of *nodes*, each of which holds a pair of processes: a *kernel* which performs the actual computation, and a *scheduler* which coordinates the created tasks. The scheduler communicates with the kernel on the same node and with the schedulers on other nodes. The *root* is that node from which the session was established. Initially, a single task runs on the root kernel; this task may create new tasks which are distributed to other kernels and may in turn create new tasks. A kernel emits messages `task:`<*t, d*> where t identifies the task and d describes it. The task needs to be forwarded to some idle kernel which eventually returns a message `result:`<*t, r*> where r represents the computed result. When a kernel emits `wait:`<*t*>, this task is blocked until the scheduler responds with the result of t. Thus, if this result is not yet available, the scheduler may submit to the now idle kernel another task; when this task has returned its result, the kernel may receive the result waited for or yet another task. A task identifier t encodes a pair <*n, i*> where n identifies the node on which the task was created and i is an index. The node n thus serves as the rendezvous point between node n' computing the result of t and node n'' requesting this result.

3. Formerly Achieved Fault Tolerance

There exist various fundamental mechanisms to achieve fault tolerance in distributed systems [1, 2, 4]. The only mechanism originally available in Distributed Maple for handling faults was a watchdog mechanism on every node that regularly checked whether a message had been recently received from the root. If not, it sent a `ping:<>` message that had to be acknowledged. If no acknowledgement arrived from the root within a certain time bound, the node shut itself down.

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

Logging Results The logging mechanism can store a consistent state of a session and is able to recover this stored state after a failure [6]. This mechanism is based on the scheduling mechanism originally available in Distributed Maple, which allows the root to receive all task descriptions created by all nodes. Before the root schedules a `task:`<*t, d*>, it saves d. When a node sends a `result:`<*t ,r*> to some other node different from the root, it forwards a copy in a `store:`<*t, r*> to the root which saves it. In a later session, the root recovers the stored data, whenever they are needed.

Tolerating Node Failures The system is capable to cope with node failures without overall failure [6]. This mechanism is based on the original watchdog mechanism. It sends a `ping:<>` to a node and supposes its failure, if it does not receive any reply in a limited time period. If the root observes that a node become unreachable, the root declares it as *dead*. The "Tolerating Node Failures" mechanism is based on the "Logging Results" mechanism, because all results that have been stored on the dead node are recovered and provided by the root. The root also recovers and reschedules to another node those task descriptors that were under processing on the dead node.

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

4. Reconnection after Connection Failure

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

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

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

For resending the corresponding wait messages, every node can use a set W_j which is used and maintained in Distributed Maple. W_j contains all wait messages sent to j; for a subset W_j^r the answers with results have already arrived from j. After j reconnected, each wait message in $W_j - W_j^r$ has to be sent again.

For resending the corresponding task, result, and store messages, every node maintains another set M_j, which contains all task, result and store messages sent to j (on the root, this set contains also those result messages which were sent by any other node to j through the root); for a subset M_j^a acknowledgements have already arrived from j. 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` message (such a message is also sent in the original case, see Section 3). If an acknowledgement arrives with index k, every element of $M_{j,k}$ is added to M_j^a. After j has been reconnected, each message in $M_j - M_j^a$ has to be sent again.

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

5. Restarting after Node Failure

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

6. Tolerating Root Failure

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

At no time during the execution of the system, there may exist more than one root. To guarantee this, the current root always has to be connected to $n/2$ nodes at least; 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, where n is the initial number of the non-root nodes.

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

At the initialization of the session, all nodes get a list about those non-root nodes which can access the storage system. These nodes are called shadow roots or simply shadows and their list is called *shadow_roots*. This list is ordered by initialization sequence of the nodes.

6.1 How a Shadow Root Becomes the Root

When the root observes that a node has become unreachable and the reconnection is unsuccessful, the root checks how many nodes remain connected. If less than $n/2$ nodes are connected, the root aborts.

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 first live node s in the *shadow_roots* list. When s receives a `root_lost:<>` message, it adds the identifier of the sender to a set R. If the root does not become unreachable for node s after a `root_lost:<>` message was received from k and k is declared *dead* by the root, s sends back an `abort:<>` message to k. Due to this message, k aborts.

If the root becomes unreachable to a shadow node s and the reconnection time expires, s 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. 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 shadow node s. Otherwise, it sends back a `root_lost:<>` message to s. Node s waits for the acknowledgements of the `check_root` broadcast for a limited time period and counts them. If the sum is less than $n/2$ where n is the initial number of the non-root nodes, s aborts. 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 the set R. (If s observes a node whose identifier is in R became unreachable, s deletes the corresponding identifier from R.) When s received a `root_lost` or a `root_connected` message from each acknowledged node (if s observes that some nodes become unreachable, it does not wait for their answers) and if the number of the elements of R is equal to or greater than $n/2$, it sends a `new_root:<>` message to all acknowledged nodes (otherwise, the node s aborts). After these nodes received this message from s, they accept node s as the new root independently of the old root, which may be reachable for some of them.

6.2 Generalization of the Node Behavior

Originally, the *shadow_roots* list has consisted of only one element. Now, we generalize the mechanism such that the *shadow_roots* list may contain more than one element.

If the root is unreachable to a node k and the reconnection time expires but the first live node in the *shadow_roots* has become also unreachable, k sends a `root_lost` message to the next live node in this list. 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 new root election. In the worst case, neither shadow node becomes the root and the whole system aborts.

6.3 After the Election of the New Root

After a shadow root has become the root and has sent the `new_root` message, 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 3). After a node has accepted a `new_root` message, it resends all store and task messages to the new root which are not acknowledged by the old root (task messages are acknowledged by the root if and only if they have already been logged). It also sends the identifiers of those tasks in a `scheduled_tasks:`<*task_identifiers*> message to the new root which are under processing on this node. The new root keeps these pieces of information in the *scheduled_tasks* table as tuples of a node identifier and a task identifier (this table is the same which are already used by the root in the "Tolerating Node Failures" mechanism in Section 3). If the new root receives a `task:`<*t, d*> message, the logging mechanism tries to restore the result of the task from *d*. If it does not manage, it checks whether *d* may be already logged with a different identifier *t'*. If yes, it checks whether *t'* occurs in the *scheduled_tasks* table. If *t'* occurs in this table, the new root drops this message, because it is already under processing somewhere. Otherwise, it schedules this task to a node.

7. Conclusion

We have implemented in Distributed Maple some fault tolerance mechanisms such that we can guarantee the following: if the system fails, we can restart it and continue the computation from a saved state; the system does not deadlock and continues normal operation, if the root crashes, some non-root node crashes, or some connection between any two nodes breaks.

There remains only one kind of *Stop* failure situation, which may let 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 on demand.

References

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

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

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

[4] 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.

[5] Wolfgang Schreiner, Christian 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 175-184, Balatonfüred, Lake Balaton, Hungary, September 10-13, 2000. Kluwer Academic Publishers, Boston.

[6] Wolfgang Schreiner, Karoly Bosa, Gabor 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.