Task Logging, Rescheduling and Peer Checking in Distributed Maple

Károly Bósa   Wolfgang Schreiner

March 18, 2002

Abstract

We have extended the parallel computer algebra environment Distributed Maple by fault tolerance mechanisms such that the time spent in a long running computation is not any wasted by the eventual occurrence of session failure. The first mechanism is the logging of task return values and of shared object values such that after a failure the newly started session can (transparently to the application program) reuse already computed result. The second mechanism is the migration of tasks such that a session may tolerate the failure of individual nodes without overall failure. The third mechanism is the redirection of the messages such that a session may tolerate also the failure of the connections between nodes without overall failure.

1 Introduction

Distributed Maple is a system for writing parallel programs in the computer algebra system Maple [3]. The system consists of two layers: The *scheduler* layer, which is a Java class library, coordinates the interactions of the *computing* layer which contains some Maple kernels. We have employed this system successfully for the parallelization of various methods and applications in algebraic geometry [5, 6]. The system is portable and its performance is analyzed in most kinds of computer networks [4], 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.

There exist various fundamental mechanisms to achieve fault tolerance in distributed systems [1, 2]. The only mechanism originally available in Distributed Maple for handling faults is a watchdog mechanism on every node that regularly checks the connection to the root node; if the root cannot be contacted any more, the node shuts itself down. As the problems became larger and larger, whose solution required more and more time, the meantime between session failures became a limited factor in the applicability of the system. Our primary goal is to extend the scheduling layer to provide fault tolerance, which can handle this problem. For this, we have to introduce a transaction or logging mechanism to the Distributed Maple which can store a consistent state of the sessions and is able to recover this stored state after a session failure [7]. The big challenge is to find a way how we can minimized the losing of the data and how we can handle efficiently the saving and recovery of data. There are numerous possibilities for faults, but we have to be sure, the system is able to continue a suspended or broken computation in each case. For achieving this, we build in several control mechanisms to the system.

Our secondary goal is to avoid the session failure if some nodes become unreachable. The system has to capable to cope with the node failures without overall failure. This mechanism is based on principle of the original watchdog mechanism. It supposes a node failure, if it does not receive any direct reply from the checked node in a limited time period, after it sent a ping message directly to this node.

The remainder of this paper is organized as follows: Section 2 collects and analyses the possible failure situations in Distributed Maple. Section 3 explains the logging and failure recovery mechanisms that we are introduced to reuse already computed results. Section 4 and Section 5 extend this by two new mechanisms. The one of them described in Section 4 allows the sessions to continue operation even if client nodes become unreachable. The other described in Section 5 does the same thing even if peer connections (connection between two clients) fail. The Section 6 summarizes our results and outlines further development directions.

2 Failure Analysis

Numerous possibly faults may occur during the execution in Distributed Maple and all of them can be ordered into the following three groups:

First such a group is the *Message failures* which covers those situations in which some messages are simply lost or duplicated or become corrupted.

The next group is called *Byzantine failures* which covers those failure situations in which some nodes behave arbitrary.

The third failure group is called *Stop failures.* This covers those failure situations in which some system components stops abnormally without any warning. In more details, we can speak about the following error situations within *Stop failures*:

- the root node crashes,
- some client nodes crash,
- some Maple processes fail,
- some connections between the root and the clients break and
- some connections between two clients break.

In the original Distributed Maple, we have to restart the computation in every such a case. If the root node crashes, the system will abort, but in all other cases it will deadlock and it should be stopped manually. We say, a node crashes if the machine crashes or reboots, or the the process in the scheduling layer crashes for some reason (a bug in the implementation of the Java Virtual Machine, of the Java Development Kit or possibly also of the Distributed Maple Software).

We have not worked on the problem of *Message* and *Byzantine failures* yet and we always assume in this paper that such failure situations never occur. We focus on the third failure group and we give solutions to some mentioned *Stop failure* problems.

3 The Logging Mechanism

The logging mechanism in Distributed Maple is a fault tolerance mechanism for saving the results of intermediate tasks and the values of shared objects during the computation. It allows to restore the results of computed tasks in a later session, if the current session crashes. The system supports two kinds of logging modes: *Fast Mode* and *Safe Mode* (see the difference in Section 3.1).

The logging mechanism can be switched on and off from Maple by the *dist.maple* interface: in the command

$$\texttt{dist}[\texttt{logging}](mode);$$

the parameter *mode* can be 0, 1 or 2, where

- 0 ... switch off the logging,
- 1 ... switch on the *Fast Mode* (see Section 3.2),
- 2 ... switch on the *Safe Mode* (see Section 3.3).

If the user switches one of logging modes on, the system begins to save the results of the intermediate tasks to a *logging directory* and also begins to restore the results of computed tasks from this directory. When the computation finishes and `dist[terminate]` command is executed, the *logging directory* is removed.

3.1 Assumptions and Guarantees

For the correct working of the logging mechanism, we have to introduce some conditions about which we assume that they are satisfied by the component of the system.

- We assume first that the Distributed Maple can access to a reliable and stable storage system, which never loses data and always works during that time, while the logging mechanism is switched on.
- We assume that the communication is reliable between the system components. Messages are never corrupted or duplicated and they may be lost only in *Stop failure* situations.
- We assume also that the system component work correctly (No *Byzantine failures*).
- At last, we assume that *Stop failures* may occur.

If the system satisfies these conditions, we can provide two kind of logging and recovery mechanisms, which are called:

Fast Mode is the base logging mechanism, by which we guarantee, if the system fails, and if the recovery mechanism does not deadlock, then we can restart it and we can continue the computation from a saved state.

Safe Mode is the advanced logging mechanism, into which we integrated some additional mechanisms. By this, we guarantee, if the system fails, we can restart it and we can continue the computation from a saved state.

Although, by these logging mechanisms, we can continue and finish a launched computation, but any additional *Stop failure* situations are not covered by them. Namely, the same failure situations may occur by using them like in Distributed Maple originally. This means in more details:

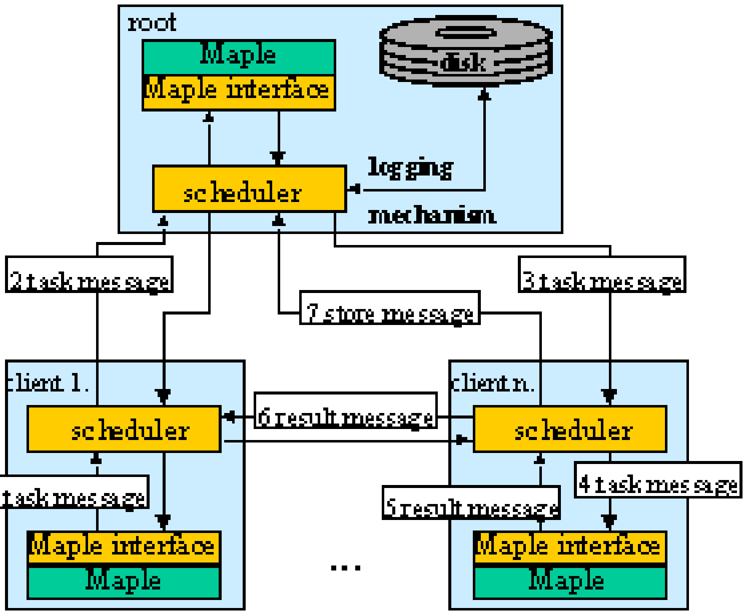

Figure 1: Execution Model

- The system aborts
  - if the root node crashes.
- The system deadlocks
  - if some client nodes crash,
  - if some Maple processes fail,
  - if some connections between the root and the clients break and
  - if some connections between two clients break.

In all such cases, when the system deadlocks, we have to force the system aborting by killing the processes manually.

3.2 Fast Mode

3.2.1 Overview

The *Fast Mode* (and the *Safe Mode* too) is based on the centralized scheduling mechanism originally available in Distributed Maple, which allows the root to receive all tasks description created by all nodes. Namely, before the root schedules a task description in one of logging modes, it tries to restore the result of the task from the hard disk (because the result may be computed and stored in a previous session). For this, the task result is identified on the hard disk by a hash code created from the task description. If it cannot restore, it saves the description and schedules the task. When the result is computed the root receives it in a *store message* and saves it to the hard disk (see Figure 1). During the implementation of this mechanism, we took care of the following cases:

- 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 it may be corrupted.
- The result of a task may not be stored yet or it is stored but it may be corrupted.
- The restoring of the result is successful, hence, the task will be not scheduled.

We assume that a task description or a task result may contain some identifiers of another task. In this case, it may occur, that a *wait message* or a *request message* refers to a task identifier which was created in a previous session. Therefore, every such a message is sent to the root, which tries to restore the corresponding result. For this, the task result is identified on the hard disk by the task identifier. 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 a more efficient way, if we use the values in the memory instead of restoring them again).
- The restoring of the result by task identifier is successful.
- The correct result is not available on the hard disk, hence, the root tries to restore the description of the task by task identifier in order to schedule it and recompute the result.
- The neither correct result nor description of a task may be available (In *Fast Mode*, the program will consequently deadlock. In *Safe Mode*, this situation cannot occur.)

We handled also that situation, when a task result is restored in a session by a hash code created from the description of the task (in this case, the current task identifier is different from the previous one), therefore the new task identifier will not be stored along with the task result to the hard disk. And, in a latter session, the root may try to restore this task result again, but it may refer to such a new task identifier.

3.2.2 Data Structures

Every task is uniquely identified by its **task identifier** which consists of 3 parts:

- A *session identifier* (see below).
- A *node identifier* says which node created the task.
- A *task index* enumerates all tasks created by the node.

The current session is uniquely identified by the **session identifier** which is part of the identifier of every task and 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 Section 3.2.3).

By the **logging message**, the mode of the logging mechanism can be changed. It has only one argument, which is the identifier of such a mode (switch on *Fast Mode*,switch on *Safe Mode*, switch off logging).
When the logging mechanism is switched on or off by the `dist[logging]` command, the root gets such a message which it sends further to every client, such that the clients can change also their working with respect to the new mode.

By the **store message**, a computed task result is sent to the root, such that the result will be saved to the hard disk. The message has two arguments, the identifier of a task and the result of this task (see Section 3.2.4).

3.2.3 Files

The logging mechanism saves all necessary data into a *logging* directory using the following files:

sessionid file 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 hard disk. If it exists, it reads and increments it. If it does not exist, the current session identifier is set to 0. If the `dist[logging](1)` command is issued (this command switches on the *Fast Mode*) and the root gets a *logging message*, then the root saves the current session identifier into this file. If the *logging* directory or *sessionid* file does not exist, they are created.

result.<*task id*> file contains the result of the task identified by the file extension. The system can restore the result of a particular task, if it knows the identifier of the task.

descr.<*task id*> file contains the description of the task identified by the file extension. The system can restore the description of a particular task, if it knows the identifier of the task. This operation is needed, if the result of the task could not be saved before the system crashed in a previous session and the system would like to restore the description of a task to restart its execution (see below).

taskid.<*hash code*> file contains the identifier of those tasks whose descriptions have the hash code denoted by the file extension. When the system wants to check whether a result has already been stored for a particular task description, it generates a hash code from the description and looks for a *taskid* file whose extension equals this hash code. If it finds it, it compares the description with the contents of those *descr* files, whose extensions are contained in this *taskid* file (if a particular *descr* file does not exist or its content is corrupted or

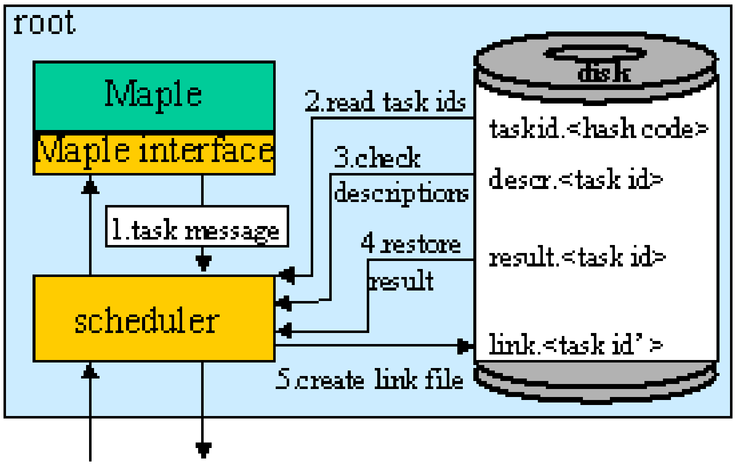

Figure 2: Restoring Results by Description

not equal to the description, the system ignores this file). If it does not find the matching *descr* file, it cannot restore the result (which may only occur if the logging session was executed in fast mode). If it finds a *descr* file whose content equals the description, then the system looks for a *result* file with the same extension. If this file does not exist or its content is corrupted, then the system cannot restore the result. Otherwise, the system restores it and saves the identifier of the stored task to a *link* file (see Figure 2).

link.<*task id*> file contains the identifier of a task whose description is the same like the description of the task identified by the file extension. The system does not create the *descr* file and the *result* file for a task description more then once, but it creates this *link* file for each latter task, which contains identifier of the first logged task. This file is needed, if the system wants to restore a task by its identifier, which has no *descr* file and *result* file. (see Section 3.2.4).

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.

3.2.4 Handling of the Tasks

The following activities which handle the tasks are changed in *Fast Mode*:

When the `dist[start]` or `dist[process]` commands are executed, the current scheduler gets a *task message* from a Maple kernel. If this scheduler is not the root, the *task message* is sent to the root. If the root gets a *task message*, it checks whether it can restore the result by the description from the hard disk. If not, the root saves the description to the hard disk by a separate thread and schedules the task (see Figure 3). Otherwise, the root restores the result of the

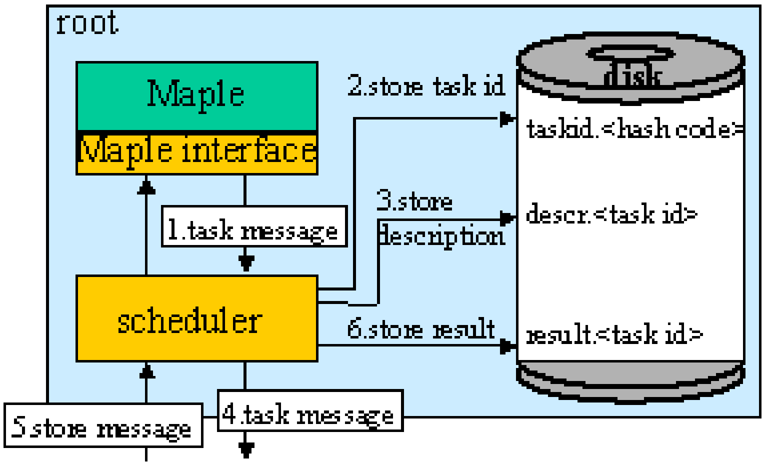

Figure 3: Storing Descriptions and Results

task and sends it in a *result message* to the node which created this task (the identifier of this node can be determined from the task identifier).

If a node has computed the result of a task, it sends it in a *result message* to the node which created this task and also sends a duplicate in a *store message* to the root. When the root gets it, it saves the result to the hard disk in a separate thread without file buffering (see Figure 3).

If a `dist[wait]` command is executed whose argument refers to a task which was created by a previous session, the command asks for the result from the root. When the root gets a request, looks for this result in the result table. If it is there, the root returns it in a result message to the node which sent the request. If it is not there, the system tries to restore the result from the hard disk (if the result cannot be restored from the *result* file, the system tries to restore the result from another *result* file identified by contents of the *link* file). If it is successful, the root sends back the result in a result message to the node which sent the request. Otherwise, the system tries to restore the description from the hard disk by the task identifier. If it is successful, the root schedules the task for execution (see Figure 4).

If it does not manage to restore the description, the system treats this identifier as that of a shared object which has not got any value yet, the program will consequently deadlock. This error cannot occur when the logging was performed in *Safe Mode*.

3.2.5 Handling of the Shared Objects

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 *Fast Mode*:

When the `dist[put]` command has been executed, the scheduler gets a *put message* from the Maple kernel. The scheduler sends the value which was in the message in a *store message* to the root.

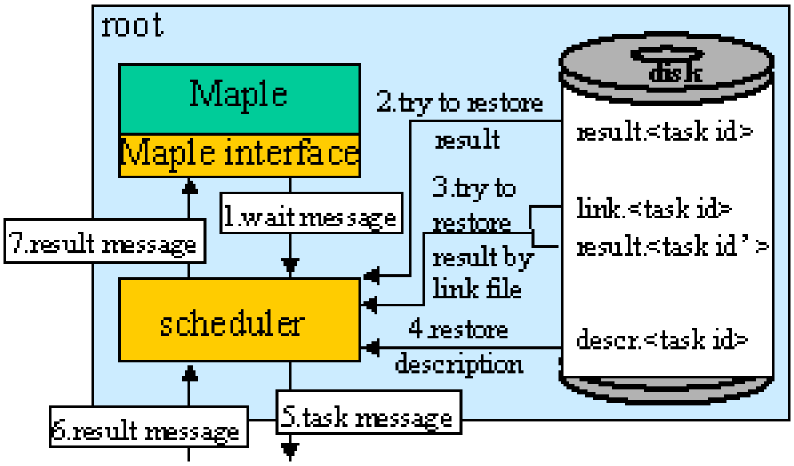

Figure 4: Restoring Results by Task Identifier

When the `dist[get]` command has been executed, the scheduler gets a *wait message* from the kernel. In this case, the activity of the scheduler is the same like for waiting for the results of tasks (see Section 3.2.4).

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.

3.3 Safe Mode

3.3.1 Overview

The *Safe Mode* behaves like the *Fast Mode* described in the previous section but it has some additional mechanisms. Namely, in the Fast Mode, if the result of a particular task was not saved in the previous session and neither was its description, the system will not restore its result (treating the identifier as that of shared object) which subsequently yields a program deadlock. The *Safe Mode* solves this problem such that this situation never occurs, because this mode guarantees that each task description is always logged, before some task descriptions or results may contain its identifier. Therefore, the restoring of a saved task result by its identifier always work correctly. For achieving this, during the implementation of the *Safe Mode*, we took care of the following three cases:

- A task description may be written to the hard disk if and only if all previous task descriptions created by the same node have been completely written. Otherwise, the current task description might contain the identifier of the previous task which was created by the same node. If the description of the current task would be written earlier than the description of the previous one and the system crashed during the writing of the description and the result of the previous task, the system would refer to this task in the next session without having logged its description or result.

- A task description may be scheduled to a node if and only if the task description has been completely written. Otherwise, this task might create some other tasks whose descriptions contain its task identifier. If these task descriptions were be written earlier than the description of their parent task and the system crashed during the writing of task description and result then the system would refer to this task in the next session without having logged its description or result.

- The result of a finished task may be sent to another node if and only if all the task descriptions which are created by this task have been completely written. Otherwise, the result of the finished task might contain the identifier of some created tasks. If an other task got such an identifier before the owner's description of the identifier were completely written, the same situation like in the previous two paragraphs would occur.

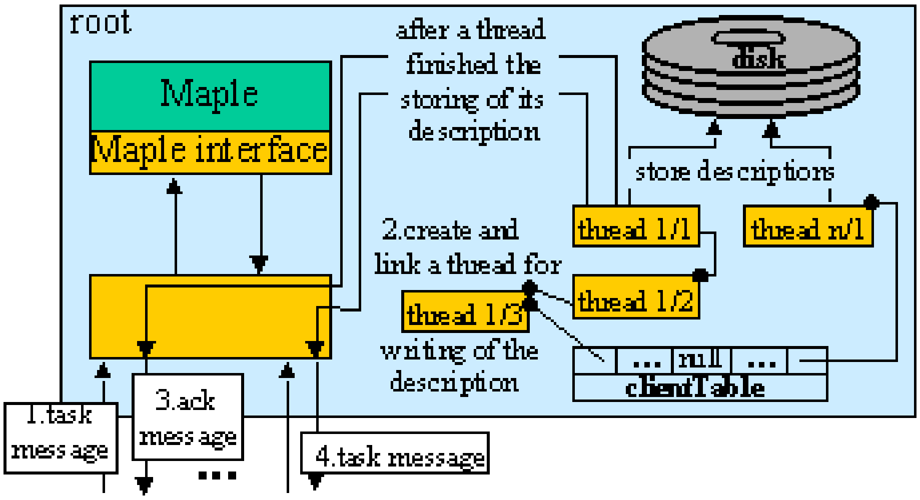

Figure 5: Storing Descriptions in Safe Mode

3.3.2 Data Structures

clientTable is a hash table whose keys are *node identifiers* and whose elements are references to threads. This data structure is located only in the root. When the root wants to save a task description to the hard disk, it creates a thread for doing this, but it does not start it immediately. First it checks whether the identifier of the node which created this task occurs as a key in this hash table. If it does not occur, it means there is no description from this node currently being logged. Therefore, the root puts the reference to the thread with the node identifier as a key into this hash table and starts the thread. Before a thread finishes, it checks whether another thread from the same node waits for a logging and, if yes, starts it. Otherwise, it removes its entry from the hash table. If such an entry occurs in this hash table, the thread has to wait until the predecessor threads have finished their execution. Therefore, the root replaces the hash table entry by a reference to the new thread and creates a link from the previous thread to new thread. Thus the previous thread can start the new one, before it finishes its execution (see Figure 5).

running_tasks is a list which contains the identifiers of all the running tasks which run on the primary Maple kernel of a node at the same time (when a running task executes a `dist[wait]` command, 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 task of a task (see Section 3.3.3).

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 Section 3.3.3).

created_tasks is a hash table whose keys are identifiers of those tasks which run on the current node (on the primary Maple kernel or on the external Maple kernels) or have already finished their execution, but have some children tasks whose descriptions have not been saved by the logging mechanism yet. An element of this table consists of a *taskids_list* queue which contains the identifiers of all children of the corresponding task. When a task has finished its execution, the scheduler does not send further its result, until the corresponding row of this hash table contains an empty *taskids_list* queue (see Section 3.3.3).

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.

result_of_the_finished_tasks is a hash table whose keys are identifiers of those tasks which have already finished, but they have some children 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 Section 3.3.3).

After a thread saved a task description, it sends an **ack message** to the node which created this task. It has one argument which is the identifier of the saved task (see Section 3.3.3).

3.3.3 Handling of the Tasks

The following activities which handle the tasks are changed in *Safe Mode*:

When the `dist[start]` or `dist[process]` commands are executed, the current scheduler gets a *task message* from a maple kernel. First, it determines the identifier of the parent task of the task by the *running_tasks* list or by the *externUsed* hash table (depending on whether the task message came from the primary Maple kernel or from an external Maple kernel). Using the parent task identifier as a key, the scheduler adds the identifier of the task as an element to

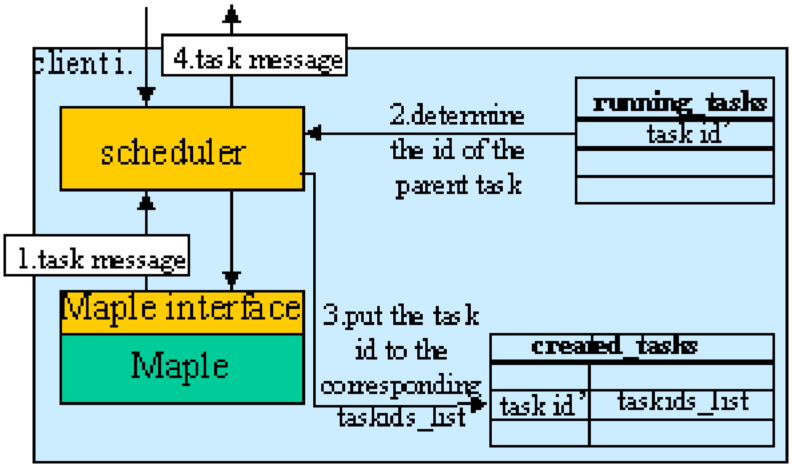

Figure 6: Determining the Parent of the new Tasks

a *taskids_list* queue in the corresponding entry of the *created_tasks* hash table. Afterwards, if this scheduler is not the root, the task message is sent to the root (see Figure 6).

If the root restores the result of the task, it sends it in a *result message* to the node which created this task and also sends an *ack message* to the same node. If the root can not restore the result, it saves the description of the task to the hard disk and only afterwards schedules the task for execution sending an *ack message* to the node which created this task.

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

When a Maple kernel sends a result in a result message to the current scheduler, the scheduler removes the identifier of the active task from the *running_tasks* list or removes the corresponding row from the *externUsed* hash table (depending on whether the result message came from the primary Maple kernel or from an external Maple kernel). Using the task identifier as a key, the scheduler checks whether the corresponding *taskids_list* queue is empty in the *created_tasks* hash table. If it is empty, the system is allowed to use this result and removes this row from the *created_tasks* hash table. If it is not empty, the system is not allowed to use this result until this *taskids_list* queue has become empty. Therefore, the scheduler puts the result with the task identifier as a key to the *result_of_the_finished_tasks* hash table.

When a scheduler gets a *ack message*, it takes the task identifier which was in the message and begins to look for this in the *created_tasks* hash table. It compares this task identifier with the first element of every *taskids_list* queue (it is enough to check the first elements because 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 the task identifier, it removes it from the corresponding *taskids_list* queue. If this queue becomes empty, the scheduler removes the entry from the *created_tasks* (whose key is the identifier of the parent task). The scheduler checks whether there exists a result for this parent task in the *result_of_the_finished_tasks* hash table. If the result exists, it removes the corresponding rows from both *created_tasks* and *result_of_the_finished_tasks* hash tables and the system begins to use the result.

3.3.4 Handling of the Shared Objects

The activities which handle the shared objects are not changed in *Safe mode* as compared to *Fast Mode.*

4 The Rescheduling of Tasks

The task rescheduling in Distributed Maple is a fault tolerance mechanism, by which the session may tolerate the failure of the clients without overall failure.

The rescheduling mechanism can be switched on from Maple by the *dist.maple* interface: in the command

$$\texttt{dist[logging]}(mode);$$

the parameter *mode* can be 3 too, where

- 3 ...switch on the rescheduling.

If the user switches the rescheduling mode on, the system automatically switches the *Safe Mode* on, because the rescheduling uses the stored data of the logging mechanism.

4.1 Assumption and Guarantees

For using the task rescheduling, we assume, that the *Safe Mode* is switched on and works correctly. Namely, if a client becomes unreachable, the root restores the corresponding logged task descriptions from the hard disk in order to reschedule them. (The *Safe Mode* guarantees, that every task description is saved to the hard disk, before it is scheduled.)
This means, we also assume all those things which we assumed and described for the correct working of the logging mechanism in Section 3.1: *reliable storage system, reliable communication, no Byzantine failures* and *only Stop failures may occur.* And this assumptions are enough for the correct working of the task rescheduling.

If the system satisfies these conditions. we guarantee

- all guarantees of the *Safe Mode* and
- if some client nodes crash or some connections between the root and the clients break, then the system continues the normal operation (by rescheduling of the corresponding tasks and resending of the corresponding messages).

This means, we managed to reduce the number of the *Stop failure* situations. In more details:

- The system aborts
  - if the root node crashes.
- The system continues the normal operation
  - if some client nodes crash and
  - if some connections between the root and the clients break.
- The system still deadlocks
  - if some Maple processes fail and
  - if some connections between two clients break.

It is already better than in Section 3.1, but two kind of error situations still have remained in which the system deadlocks.

4.2 Overview

The task rescheduling mechanism is based on the watchdog mechanism originally available in Distributed Maple. This means, the root checks periodically all client. If a client does not send any message to the root during a limited time period, the root thinks, the client became unreachable. This watchdog mechanism cannot make a distinction between the following situations:

- A client crashes.
- The connection between the root and a client breaks.
- The answer from a client 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 client, but the client is still alive, the client will abort after some time. When the root observes that a client became unreachable, it reschedules those tasks which were scheduled on the client and sends a broadcast to all other client. Due to this broadcast, all nodes (the root and all clients) resend those messages to the root which might be lost. Namely, the task results which were located on the unreachable client are restored and provided by the root.

4.3 Data Structures

crash is a hash table whose keys are client identifiers and whose elements are boolean values which show the states of the clients. If a value is false, the corresponding client lives. If a value is true, the corresponding client became unreachable.

sockets is a hash table whose keys are client identifiers and whose elements are references to those sockets, via which the root or a client interacts with other clients (see Section 4.4).

inThreads is a hash table whose keys are client identifiers and whose elements are references of input threads which handle the arriving messages from the corresponding clients (see Section 4.4).

outThreads is a hash table whose keys are client identifiers and whose elements are references of output threads which send the messages from a message queue to the corresponding client.

scheduled_tasks is a hash table whose keys are node identifiers and whose elements are task identifiers which are scheduled to the corresponding node. This data structure is located only in the root.

The new version of the **ping message** has an argument which is the identifier of a node which would get this message.

dead message has an argument. This message is sent if the watchdog of the root observes that one of clients became unreachable. The argument is the identifier of the unreachable client (see Section 4.4).

4.4 Activities

The following activities are changed in the *tasks rescheduling* mechanism:

When the root gets a *start message* or a *startp message* and it creates a new client, it puts the references of the socket, the input thread and the output thread of the new client to the *sockets*, *inThreads* and *outThreads* hash tables. The root also inserts the client identifier with a boolean value false into the *crash* hash table.

When a client wants to send a message to another client through a peer connection, but such a connection has not existed between these two clients yet, the client creates it. Afterwards, the client puts the references of the socket, the input thread and the output thread of the new peer connection with the identifier of the remote client as a key to the *sockets*, *inThreads* and *outThreads* hash tables. It also inserts the client identifier with a boolean value false into the *crash* hash table.

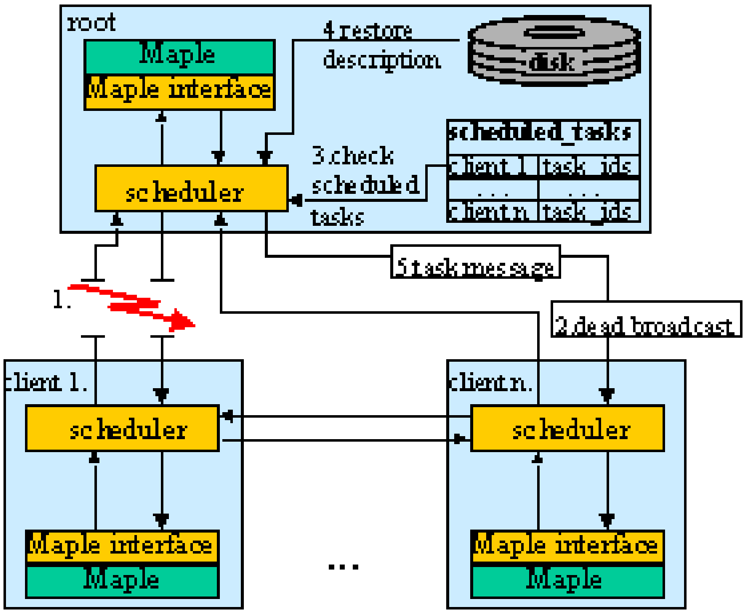

Figure 7: Rescheduling Tasks, If a Client Becomes Unreachable

The watchdog thread of the clients inherited the activity of the previous version of the watchdog. Namely, it observes the root and if the root became unreachable, it would finish the execution of the application on the client.

By the new watchdog thread of the root, the root monitors the clients. If it observes one of them became unreachable it sends the identifier of the unreachable client in a *dead message* to the main thread of the root.

Before the root schedules a task to a node, it puts the identifier of the task with the identifier of the target node as a key to the *scheduled_tasks* hash table.

When the root gets a *store message* from a client, it removes the corresponding row from the *scheduled_tasks* hash table and it checks whether any *wait messages* or *request messages* have already waited for the result which was contained in the *store message*. If it finds such a message, it sends back the result. (Remark: If the root does not find any corresponding row in the *scheduled_tasks* hash table, then the message contained a shared object.)

Before the root sends a broadcast message, it checks by the *crash* hash table, which clients live and sends the broadcast message only to those clients.

When the root gets a *dead message*, it informs the user of the peer connection failure first. Then it closes the socket of the unreachable client by the reference which it found in the *sockets* hash table and it stops the input and output threads of the unreachable client by the references which it found in the *inThreads* and *outThreads* hash tables. Then, it puts a true value with the corresponding key to the *crash* hash table and it sends a broadcast *dead message*. Afterwards, the root checks in the *scheduled_tasks* hash table which tasks were scheduled to the unreachable client, restores their description from the hard disk, and reschedules them (see Figure 7). Finally, it checks in the *externWaits* table and the *externStack* stack which *wait messages* were sent to the unreachable client and sends them to itself. These *wait messages* will get answer by *store messages*. (Remark: The *externWaits* table contains those *wait messages* which were sent from an external maple kernel. The *externStack* stack contains those ones which were sent from the primary Maple kernel.)

When a client gets a *dead message*, it checks whether it has a peer connection to the unreachable client. If yes, it closes the socket of this connection and stops the input and output threads of this connection. Afterwards, it also puts the value "true" with the corresponding key to the *crash* hash table. Finally, the client handles the *wait messages* in the same way like the root.

Before sending a result message to a client, it is checked whether the client is reported as unreachable. If yes, the message is not sent.

If a node wants to send a *request message* to a unreachable client, then it sends it to the root. When the root gets a *request message* or a *wait message* with the identifier of a task which was created by a unreachable client, it tries to restore its result from the hard disk. If this is not possible, the root will eventually receive an answer by a *store message* and then send an answer.

5 The Checking of Peer Connections

The peer connections checking in Distributed Maple is a fault tolerance mechanism, by which the session may tolerate the failure of some peer connections without overall failure. (The direct connection between two clients is called peer connection.) This mechanism is always switched on, when the task rescheduling is switched on.

5.1 Assumption and Guarantees

There is no reason for using of the peer connections checking without using the task rescheduling, because some messages may be sent via root instead of sending directly by peer connection. Hence, we assume, if some connections break between the root and some clients, then the system continues by the normal operation. This simply means, we assume, that the task rescheduling is switched on. We also assume all those things which we assumed and described for the correct working of rescheduling of tasks in Section 4.1: *logging, reliable storage system, reliable communication, no Byzantine failures* and *only Stop failures may occur*. And this assumptions are enough for the correct working of the task rescheduling.

If the system satisfies these conditions, we guarantee

- all guarantees of the logging and the task rescheduling,
- if some connections between clients break, then system continues the normal operation (by resending and redirecting of the corresponding messages).

This means, we reduced further the number of the *Stop failure* situations in Distributed Maple:

- The system aborts
  - if the root crashes.
- The system continues the normal operation
  - if some client nodes crash,
  - if some connections between the root and the clients break and
  - if some connections between two clients break.
- It remains only one kind of *Stop failure* situation, which may be the reason why the system deadlocks;
  - if some Maple processes fail.

5.2 Overview

The peer connections checking mechanism is based on the watchdog mechanism originally available in Distributed Maple. Namely, The clients check the connections between each other with the same strategy as the clients check root. If a direct connection exists between two client and one of the clients does not send any message to the other client during a limited time period, the other thinks, the peer connection to the first client broke. This watchdog mechanism cannot make a distinction between the following situations:

- The client on the other side of the connection crashes.
- The peer connection breaks.
- The answer from one client arrives too late (the network latency is longer than the limited time period which the watchdog uses).

As at the task rescheduling in Section 4, the third case is not a problem, because if one of the clients interrupts a peer connection, which is unbroken, then only some messages will be resend and if it is needed, the connection will be recreated (see below). But, in the first case, we had to analyse the situation, when a client crashes and an other client observes this earlier than the root. Then it begins to resend the messages which might be lost, because it thinks only the peer connection broke. Of course, these messages are lost. When the root observes too, what has happened, it handles the situation as described in Section 4. This means among others, the root sends a broadcast to all client about the client crashing. Due to this, the clients never try to send messages to the crashed client any more and they begin to resend the messages to the root which might be lost. This means only some redundancy may occur here, because some messages might be resend twice.

As we mentioned, if a client observes a peer connection might break, it tries to resend the corresponding messages. During the implementation of the peer connections checking mechanism, we took care of the following cases:

- In contrast to the connections between root and clients, 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 clients have to interact via the root.
- If the establishing of a peer connection was unsuccessful, the corresponding clients may try it again after some time.
- If a peer connection breaks, both clients have to become aware of this.
- The clients have to determine which messages might be lost.

5.3 Data Structures

peerConnections is a BitSet, which is located in all clients. Each bit in this data structure belongs to a possible peer connection to another client and vice versa. If a peer connection exists between two clients, then the corresponding bit is set in both client. If a client observes that a peer connection broke, the corresponding bit is cleared in the side of the client.

currentPeers is a BitSet, which is located in all clients. Each bit in this data structure belongs to a possible peer connection to another client and vice versa. A clients checks only those existing peer connections by watchdog mechanism whose bits are set in the *currentPeers*. Namely, the watchdog mechanism which runs in a thread separately is based on a limited time period. Such a time period is called a *watchdog period*. The beginning of a watchdog period, the watchdog thread synchronizes the value of *currentPeers* to value of *peerConnections*. Hence, during a watchdog period, this thread waits for messages via only those peer connections (or forces a reply via them at least) which were created in the previous period or earlier. (Each client has one period at least to send some message via a new peer connection.)
If a client observes that a peer connection broke, the corresponding bit is cleared in the side of the client.

status is a hash table whose keys are client identifiers and whose elements are integer values. If such a value is 0, messages may be sent directly to the corresponding client via peer connection. If such a integer value is greater than 0, messages are sent to the corresponding client through root. Each watchdog period decrements these values of this hash table, until they become 0.

number_of_failure is a hash table whose keys are client identifiers and whose elements are integer values which show how many times the peer connection to the corresponding client broke. If a client observes that another client becomes unreachable via peer connection, the client increments the corresponding value in this data structure. When a new peer connection is established, the corresponding values of the two clients are compared whether they equal. If they are not equal, that means one of clients not observes all the peer connection failures between this two clients (see Section 5.4).

peerbroken message has an argument. This message is sent if the watchdog of a client observes that one of clients became unreachable via peer connection. The argument is the identifier of the other client.

If a client receives a *result message* via a peer connection, it sends back an **ack_result message** to the sender client. This message has an argument which is the identifier of that task whose result was in the *result message* (see Section 5.4).

5.4 Activities

The following activities are changed in the *peer checking* mechanism:

When a client wants to send a message to another client at the first time, it tries to establish a new peer connection. If the other client does not answer for its connection request during a limited time, it sends the message to the other client through the root and it sets the corresponding value of *status* hash table to a positive number (Which means, every message whose target is the other client will be sent through the root until the corresponding value of the *status* hash table become 0 again). If the other client answers for the connection request, both clients set the corresponding bit in the *peerConnections* BitSet. Then the first client sends the corresponding value of *number_of_failure* hash table via the peer connection. It sends the message to the other client only afterwards. When the other client receives this value, it compares with the corresponding value of its own *number_of_failure* hash table. If they are not equal, it means this client did not observe the last peer connection failure and some messages might be lost. (see below).

There are two kind of messages, which should be resent in order that the system does not deadlock, if a peer connection breaks. The one of them is the *request message*. A client can resend this kind of messages which may be lost, if it checks in the *externWaits* table and the *externStack* stack which *request messages* were sent via broken peer connection (see Section 4.4). The other kind of such messages is the *result message*. If a client wants to resend such a message, it has know the sent results. Therefore, these are stored in the *result_of_the_finished_tasks* hash table (see Section 3.3.2) until the client receives back a task identifier in an *ack_result message* which identifes a result in this table.
A client sends an *ack_result message* if and only if it receives a *result message* via a peer connection.

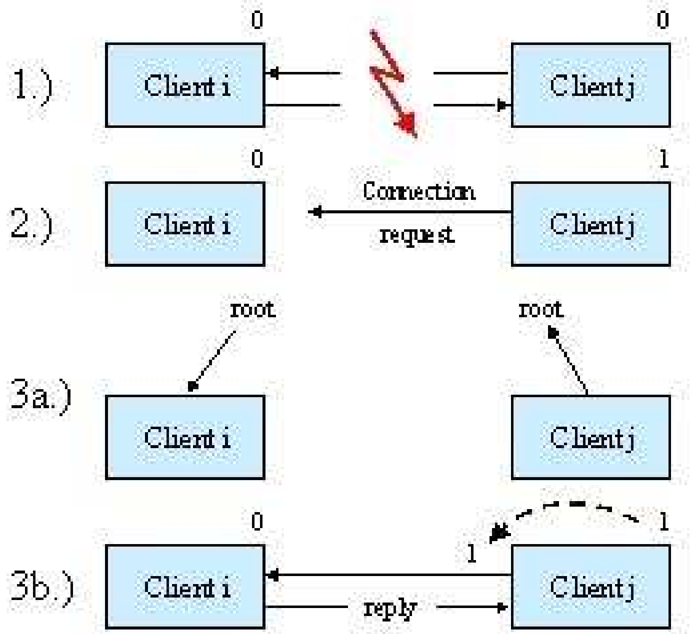

Figure 8: Recreating a Peer Connection: 1) Connection breaks 2) *Client j* observes it. Increment the value of *number_of_failure* and if it is needed try to reconnect to *client i*. 3a) If *client j* does not receive any answer from *client i*, *client j* will send to corresponding messages to *client i* trough root until some time. 3b) If *client i* replies, *client j* sends the corresponding value of *number_of_failure* to *client i* for comparing.

The new watchdog thread of the clients extends the activity of the previous version of the watchdog. Namely, all of them monitor all the existing peer connection of the client, too. If such a watchdog thread observes that another client became unreachable via peer connection, it clears the corresponding bits in *peerConnections* and *currentPeers* BitSets, increments the corresponding value in *number_of_failure* hash table and closes this side of the broken connection (closes the socket of the connection and stops the input and output threads of the connection). Finally, it sends the identifier of the unreachable client in a *peerbroken message* to the main thread of the client.

When a client gets a *peerbroken message* (from its watchdog thread), it sends it further to the root and resends those *request message* and *result message* to the other client, which might be lost.

When the root gets a *peerbroken message* (from a client), it informs the user of the peer connection failure.

If a client wants to send a message to another client, but it observed previously that the peer connection broke between them, then it checks whether the corresponding value of the *status* hash table is 0. If it is greater than 0, the message is sent through the root. If it is 0, it tries to establish a new peer connection. If the other client does not answer for its connection request during a limited time, the message is sent through the root and it sets the corresponding value of *status* hash table to a positive number. If the other client answers for the connection request, both clients set the corresponding bit in the *peerConnections* BitSet. Then the first client sends the corresponding value of *number_of_failure* hash table via the peer connection (see Figure 8). It sends the message to the other client only afterwards. When the other client receives this value, it compares with the corresponding value of its own *number_of_failure* hash table. If they are not equal, it means this client did not observe the last peer connection failure. Therefore, the connection thread which is responsible for handling of the connection requesting sends the identifier of the first client in a *peerbroken message* to the main thread of this client in this case.

6 Conclusions

We have implemented in Distributed Maple the logging and recovery mechanisms described in Section 3 such that all program sessions operating in the functional parallel programming model can be recovered. Up to now, the problem sizes that Distributed Maple could successfully deal with was limited by the meantime between failures, which was less than a day. With this new fault tolerance mechanism, we are able to tackle problems whose solution takes much longer.

When a session crashes, the system has to be restarted in order to recover the intermediate results and proceed the computation. For this, human activity is needed. We would like to improve the fault tolerance mechanisms of the system such that we can avoid these system restarting and session recovery phases in most case of the *Stop failures*. The implementation of the task rescheduling mechanism described in Section 4 and peer connections checking described in Section 5 were the first steps for achieving this goal such that sessions can continue operation even if some nodes become unreachable.

However, our implementation is not yet complete. It cannot make a distinction between the situations that a client node crashes or a connection to a client node breaks. In the second case, it would be a more efficient way to try to reconnect to this client node first, instead of starting to use the rescheduling mechanism immediately. On the other hand, there is no information about losing of the connection to a client. Namely, if a Maple kernel crashes because the corresponding machine has no enough memory for a particular computation, the root should receive some report about this reason. Otherwise, this situation would occur easily again after the corresponding tasks are rescheduled to some other machines.

More or less, the root node remains a single point of session failure. In the current version of the system, this single root node performs the task scheduling decisions and all logging activities (the model is therefore not scalable beyond a certain number of nodes). In the future, we should find a way that the system is able to substitute for the root node and sessions can continue operation even if the root node cannot be contacted anymore.

References

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

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

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

[4] 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.

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

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

[7] 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.