Published in: Proceedings of the Compcon Spring ‘93. IEEE Computer Society Press, San Francisco 1993, pp. 244-252.

Fault Tolerance Lessons Applied to Parallel Computing

Joachim Maier
IPVR
Stuttgart University
joachim.maier@informatik.uni-stuttgart.de

Abstract

This paper describes an approach to fault-tolerant parallel computing which is based on the experiences with the most successful fault-tolerant software – the transaction processing systems. The algorithms presented here have less runtime overhead and faster recovery than most preceding approaches. In the Pact parallel programming environment fault tolerance is provided fully user transparent in an easy-to-use parallel programming model with a simple synchronization model on virtual shared data.

1 Introduction

Parallel programming often fails because of its complexity which is much higher than that of sequential programming. The programmer has to deal with process-creation and synchronization, has to avoid deadlocks, and has to balance the load. If the parallel computer has distributed memory, it is necessary to get the data to the processes which need it. In fact, these are all system programming jobs which should better be done by the runtime system and the operating system.

Especially for long-running jobs, which are typical for classical supercomputer applications, it is also necessary to deal with system faults, because the higher number of components (processors, etc.) in parallel computers increases the probability of a system fault. Besides this, parallel computers usually do not run under operating systems which are used and developed since decades, but which are already under development (e.g. OSF/1 on Intel’s Paragon and on Kendal Square’s KSR1). It should be clear that the probability of operating system failures on such systems is higher than on standard systems.

But using distributed memory parallel computers is also a chance in direction of fault tolerance. If the system has clear failure semantics [5] and accepts single-node failures, it is possible to build not only a fault-tolerant programming environment, but to guarantee the “nonstop” execution of applications [2], [8].

2 Fault Tolerance

Unfortunately, most systems do not provide mechanisms to fault-tolerant execution of programs. So the programmer has to deal with faults by himself.

The easiest way to do this is by restarting the program once again after a system failure occurs. This means that all work done until the crash is lost. If this is not acceptable, and for long-running applications it is definitely not, the program has to deal with fault tolerance, e.g by using *checkpoints*.

In the absence of a checkpointing library this can be done by synchronizing all processes at a consistent global state and write the results of the computations made up to this time in a checkpoint file. If the system fails after the checkpoint is complete, the program can be restarted by the programmer from the most recently saved state.

The problem here is that the programmer has to decide which data has to be written in the checkpoint file and how to recover from this data. This is an unsatisfying solution, because all work has to be done by the programmer which increases the complexity of parallel programming once more, and besides the global synchronization, which is needed here, produces a lot of runtime overhead.

To avoid these disadvantages a checkpointing library should be used, which implements one of the numerous checkpointing protocols for parallel and distributed computer systems known in literature.

These checkpointing schemes can broadly be classified into *pessimistic* and *optimistic* schemes [13]. Pessimistic schemes as presented in [3] and [18] log each message synchronously as it is sent or received. This leads to quick recovery, but the overhead of synchronously logging every message makes these schemes prohibitively expensive for applications on parallel computers where frequent message transfers occur. These schemes are named “pessimistic", because they assume high failure probability as it is typical for distributed computing and therefore optimize recovery on the cost of high overhead during normal processing.

Optimistic schemes, such as those presented in [10] and [19] are based on asynchronous logging of messages. This causes low overhead on normal processing and allows the processes to choose the time to checkpoint their internal state and their messages. This is purchased by a complex recovery algorithm. Recovery requires global rollback to a saved state and subsequent redo of logged messages to restore a consistent global state. Hence, recovery is much more expensive than in pessimistic schemes. The name "optimistic" comes from the optimistic assumption that failures are rare.

Recent approaches like the protocols proposed in [6] and [13] try to optimize the drawbacks of optimistic checkpointing.

Process pairs as described in [2] and [4] are used e.g. on the Tandem's NonStop™ and Nixdorf's Targon™ systems to guarantee the fault tolerance of servers in an operating or database system. These servers are duplexed in an active *primary* and an inactive *backup* server with the same program and data on different nodes in different address spaces. Process pairs fit excellently with distributed memory computers if they have clear failure semantics and make them tolerant against single-node failures and transient software faults. To get a process pair in a consistent state every message must be sent to both processes. While this concept fits well into operating systems (about 15% runtime overhead as reported in [4]), for parallel programming it reduces the bandwidth and increases the latency of the message subsystem in an unacceptable way.

Besides all advantages and disadvantages of the concepts presented up to this point, the major drawback of most systems is that fault tolerance is not fully transparent to the programmer.

For database systems fault tolerance is guaranteed in a fully transparent way by using *transactions*. Transactions are atomic units of execution, which means that either the whole transaction runs successfully or, if a fault occurs during its execution, all changes on the database are undone as if the transaction has never been running. This leaves the database in a consistent state, even if a system fault occurs.

Transaction processing systems have been the subject of studies over decades in various implementations for single- and multiprocessor-systems, and for distributed systems. For a detailed description of transaction processing systems refer to [9].

Transactions, as first described in [7], are not designed for parallel programming but for database applications, where a user's program should run in an *atomic* (guarantees fault tolerance) and *isolated* (avoids multi-user anomalies) way on a shared database. All changes of the transaction have to be *durable* after the transaction's *commit* and the database should stay in a *consistent* state under all circumstances.

While atomicity and consistency are also desirable for parallel programming, isolation a priori is not desirable and durability is mostly unnecessary. This is the reason why transactions are not useful for parallel programming without any changes. Some work in this direction is described in [14] and [20] where *atomic actions* are used for the implementation of object oriented distributed systems with durable objects.

Because transactions are independent units of execution, there is no need for a control structure between them. For parallel programming there is an essential need for a flexible, easy-to-use control structure to control the program's parallel execution.

In the *Pact* (parallel actions) parallel programming environment, the approach of atomic actions as fault-tolerant units of parallel execution is used to build an easy-to-use parallel programming environment with low runtime overhead. This is described in the next few sections.

3 The Pact Programming Model

Since this is not the topic of this paper, Pact's programming model here is only described as far as it is necessary to understand the following presentation.The programming model and Pact's extensions to procedural programming languages (e.g. C) are described in more detail in [16].

Pact is based on a shared data model. Each parallel action has access to global shared data. In opposite to the shared memory paradigm used on most shared-memory multi-processors, where each parallel process a priori has free access to the whole shared memory, Pact's actions use an access-dependent concurrency control schema. This helps avoiding synchronization faults, which are one of the most frequent faults in parallel programs and usually are very hard to find.

To choose the best access to the shared data the programmer has the possibility to define *dataunits*, which are parts of the shared data (e.g. rows of a matrix for a gaussian elimination algorithm). These dataunits are the unit of concurrency control, and therefore avoid lock conflicts.

The execution order of parallel actions is defined by *events*. Each action is connected to three events: its *start-event*, its *termination-event* and its *error-event*. When the start-event has been triggered, the action is scheduled by the runtime system. If the action has successfully terminated its execution, the termination-event fires which triggers the execution of other actions. The error-event gives the programmer the possibility to react on transient software faults. A simple example for an execution plan showing only start- and termination-events in form of a dependency graph is illustrated in Fig. 1. In this example the actions g(1), g(2), and f(3) are executed in parallel after the termination of s() – the event ParBegin is triggered. The action e(x) is started after the event ParEnd is triggered. This is when the actions g(1), g(2), and f(3) are terminated.

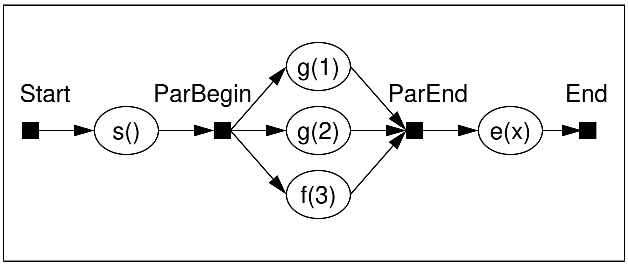

Fig. 1 Pact execution plan.

4 The Pact Runtime System

The prototype's runtime system which is currently implemented on a Tandem shared nothing multiprocessor system is a process system as illustrated in Fig. 2. The system is based on two major decisions:

- Shared data is emulated by a *virtual shared data* algorithm.
- No dynamic process creation, but preallocated *server-processes* are used.

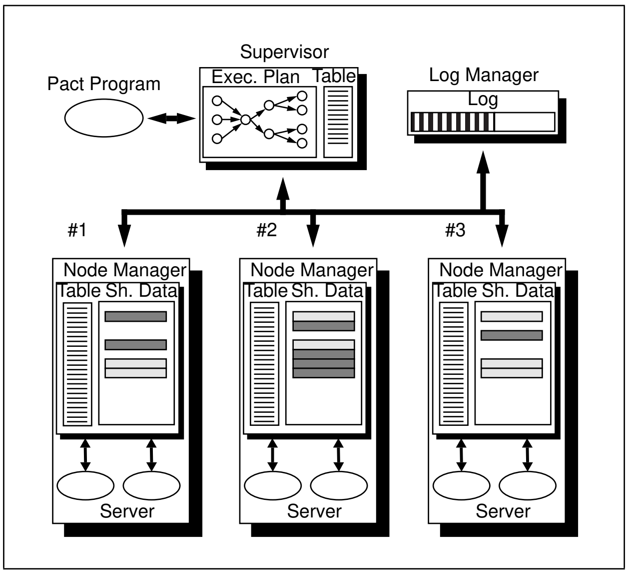

Fig. 2 The Pact runtime system

The decision to use this type of architecture is made, because in absence of *threads* [1] process creation is too expensive for a large number of dynamic process starts. Instead of this, a preallocated number of server-processes for each type of action is started at the start-up of the runtime system. During execution of the parallel program new processes are started only as part of recovery. A virtual shared data algorithm is necessary for an implementation of (virtually) shared data on a distributed-memory architecture. Note that there are more than one server process of each server type on a processor node. This is necessary because during the time a server process is waiting on a dataunit another server on this node is able to run.

If the runtime system is started successfully the execution plan which is defined in the Pact Program using actions and events is passed to the Supervisor which controls the execution on all involved processor nodes and schedules the actions to the server processes. The Node Manager controls the accesses to virtual shared data which is physically shared between the server processes on the processor. The Log Manager logs the actions in the runtime system in order to enable recovery.

The *Supervisor* which is centralized in this prototype implementation distributes the actions to the corresponding servers on the processor nodes. This is done depending on the execution plan defined by the Pact Program and the occurrence of start-events. For reasons of efficiency the Supervisor uses a data-dependent load balancing strategy. This means that it uses compiler generated or user given information about the dataunits an action wants to use, in order to decide if the action should run on the node which already has the necessary dataunits. A problem here is that depending on the virtual shared data algorithm, it is impossible for the Supervisor to have actual information about the dataunits' locations. For reasons of efficiency the Supervisor cannot get involved in every data transfer between the nodes, but it gets actual information whenever an action terminates. At this time the action server sends its results together with information on the dataunits it has used to the Supervisor in one message. Additional to this, the Supervisor's tasks are those of a recovery manager. Whenever a component of the Pact runtime system crashes, the Supervisor recovers all lost data and actions as described in more detail in section 5.

In order to recover system failures it is essential to hold an actual copy of all dataunits on the log. The *Log Manager* coordinates this logging activity. Whenever an action gets ready during its commit phase, it has to send all changed dataunits and its results to the Log Manager. When this is done successfully and all locks are released the action's results are sent to the Supervisor. If a node crashes after this commit, the Supervisor can recover the actual version of all lost data from the log. Another situation in which the log is needed is when actions get into a deadlock situation or when the action fails because of a software fault. If the system detects a deadlock, it *aborts* one of the participated actions and restores the unchanged data on the node. This is equal to a rollback of all updates which are done by the action. Although the Log Manager is centralized in this prototype, it is easy to distribute it like it is done in some database management systems.

The *Node Managers* administrate in a fully distributed way the system's virtual shared data.This is emulated by an algorithm adapted from Li's and Hudak's *dynamic distributed page ownership* algorithm [12]. For this purpose each Node Manager uses a table in which the actual *probable owner* of every dataunit is stored. In this table the locks are managed, too. Whereby each Node Manager manages exactly the locks of those dataunits which are owned by itself. If a server requests a dataunit, it sends a message to its Node Manager. The latter gives it a pointer to the dataunit in the nodes' local memory which is shared between the Node Manager and the servers on the node. If the Node Manager cannot satisfy the request from its local server, it asks the probable owner of the requested unit to give it to it. If this is not possible because of lock conflicts, the request is queued. Tandem's communication protocol which notes the sender, if the recipient of a message does not exist, makes it possible for the Node Manager to detect system failures. Whenever it discovers a failure, it informs the Supervisor which coordinates all necessary recovery activities. Note that in opposite to Li's and Hudak's algorithm user-defined *dataunits* instead of virtual memory *pages* are passed through the system. This reduces the number of messages and the message length, because the granularity of the dataunits fits exactly with the granularity of access. If the program needs only a few bytes, only these few bytes are sent to (and locked for) the server. If the program needs thousands of bytes which need many pages, the dataunit is sent in only one message with only one message latency.

For a more detailed description of the runtime system and the virtual shared data algorithm refer to [15].

5 Fault Tolerance in Pact

Fault tolerance in a transaction processing system is based on the ACID properties of transactions. Quoted from [9] these properties are:

Atomicity – a transaction's changes to the state are atomic: either all happen or none happen.

Consistency – a transaction is a correct transformation of state. The actions taken as a group do not violate any of the integrity constraints associated with the state. This requires that the transaction is a correct program.

Isolation – even though transactions execute concurrently, it appears to each transaction T, that others executed either before T or after T, but not both.

Durability – Once a transaction completes successfully (commits), its changes to the state survive failures.

Now the interesting question is, which of these properties are desirable for parallel programming, too?

Atomicity is the basic characteristic needed for fault tolerance. If the system runs correctly, nothing unusual happens to the action – it executes like an asynchronous procedure call. But if not, the effects of the action up to the failure are undone. If a fault occurs during the action's execution, it can be scheduled again (perhaps it was a transient software fault) or the program abends with an error message to the user. If a system failure occurs, the runtime system has to be recovered and the parallel program can be resumed by undoing all failed actions and do them again. Therefore, atomicity is a basic characteristic of Pact's actions.

Everybody will agree that *consistency* is necessary. But in this context it is difficult to give a definition of consistency on shared data used for parallel programming, since it depends on the application. The most obvious definition is: data is in a consistent state, if only updates of committed actions are visible on it. This definition is used currently in the Pact prototype. But think about a parallel iterative solver of a linear equation system. In this case a consistent state transformation can be defined as any transformation, which does not worsen the norm of the matrix. Therefore, in case of a failure an action's uncommitted updates must not be undone, if the norm of the matrix has been decreased.

Isolation is a property which seems absurd for parallel programming. On parallel computers almost everything is done to make communication between processor nodes (via message-passing or shared-memory access) as fast as possible – not to isolate them. But the practice in parallel programming shows that it is very difficult to write correct parallel programs with irregular access to shared data (by message-passing or by shared-memory). Mostly if the type of access is not read-only this access has to be coordinated by a very restrictive policy (e.g. monitors). So the easiest way to avoid lost updates and dirty reads is to use an access-dependent two-phase locking protocol. The advantage for fault tolerance is, if actions are isolated against each others, then there is no need for undoing other actions than the failed ones, because every action only sees committed changes. This helps avoiding the domino effect at recovery. The drawbacks of using lock protocols are the administration overhead and the danger of reducing parallelism by lock contention. But these disadvantages are reduced here, because lock management is included in virtual data management and the analysis of typical supercomputer applications show that lock conflicts are rare. In fact, isolation depends on the definition of consistency and so it is possible to use other types of concurrency control mechanisms depending on a programmer-defined consistency definition.

Durability as used in transaction processing systems guarantees that if an user has seen the commit of a transaction, he can be sure that all changes on the database are durable. This type of durability is not needed for parallel programming. It is just necessary to recover the program and do all failed or undone actions again. But there are some restrictions. Actions in parallel programming are not independent, but the termination of an action can trigger other actions. Besides this, for reasons of efficiency undo and redo of actions at recovery should be limited to those which are absolutely necessary. These are the actions which are active on failed nodes at the moment of the failure. In Pact this is guaranteed by a force-log-at-commit protocol. But there is no real need for making shared-data non-volatile as some approaches to fault-tolerant parallel computing do.

All these properties are guaranteed by the Pact runtime system with the two-phase locking protocol and the following commit protocol.

5.1 Commit

Every action has exactly two possibilities to terminate. If no error or system-fault occurs either it commits or it aborts. In the *commit* case the actions' results and side-effects (updates on shared-data) are logged and the locks are released.This enables redo-recovery. After the action has committed the action's termination event is triggered.

If the action *aborts*, the locks are released and all updates are undone. If the abort is the result of a program error, undo of updates on shared-data with the help of data from the log-manager is required. If the abort is triggered by a system-fault, no undo is required, because the data on this node is lost. If an error occurs and an error-event is defined to the action, it is triggered, but if a system-fault happens the action is redone.

Now lets have a closer look to the commit protocol.

1. The server sends a synchronous *commit message* to its Node Manager. This includes the action's reference parameters (results) and the identifiers of updated dataunits.
2. The Node Manager sends the action's reference parameters and the updated dataunits asynchronously to the Log Manager (the *log message*) in order to serve requests of other servers and Node Managers (for virtual shared-data management and commit coordination). When the Log Manager acknowledges the receipt of the commit message by an *acknowledge message* after writing it to the log, the action is committed logically. From now on the action is recoverable.
3. The action's locks are released and queued requests on freed dataunits are handled.
4. The Node Manager sends a *committed message* to the Supervisor. This message includes the action's results and the identifiers of the updated dataunits. By using this information the Supervisor's dataunit ownership table is updated, which is used for load-balancing. The results of the action are forwarded to the Pact Program by the Supervisor.

The message-flow for this protocol is illustrated in Fig. 3 where synchronous messages are illustrated as black arrows and asynchronous messages as gray arrows.

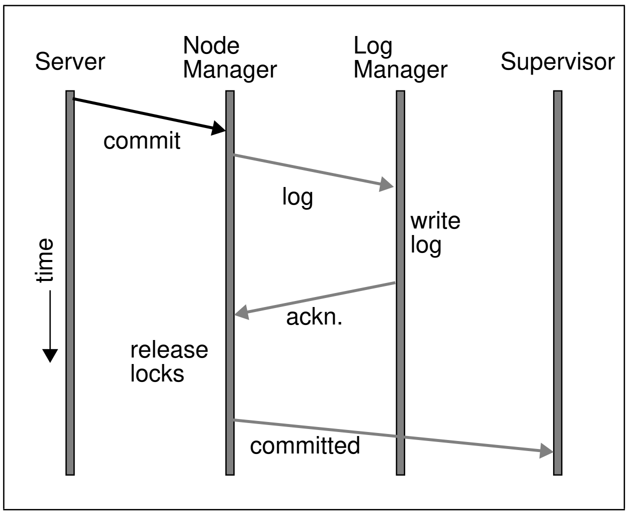

Fig. 3 Message-flow for commit

5.2 Log Management

The implementation policy of the Log Manager depends on the question which types of failures should be recoverable. If it is not necessary to recover complete system crashes and failures of more than one node during the execution, because the possibility of such failures is low, then there is no need to write the log on persistent memory. This makes logging faster, because the delay of logging at commit is not much higher than the time for two messages. The data-structures needed for this type of Log Manager is a index-structure for all of the system's shared dataunits and a list for all result-parameters of committed actions. This allows undos for aborts and recovery of dataunits lost by single-node failures. This Log Manager should run on another node than all other Pact processes to make the system tolerant against a single-node failure. But if the node where the Log Manager resides fails, the system can no longer abort any actions and so has to stop if an error occurs during an action.

Another problem with this design is that the log data-structure is held in virtual memory. If the log data does not fit into real memory – and usually it does not – writing on the non-persistent log produces disk I/O, too (page-faults). Therefore, it is better to choose a persistent Log Manager design.

This persistent Log Manager is a process which during normal operation does nothing else but writes the received log data as fast as possible into a sequential log file and after this sends back the acknowledge message. To achieve highest performance the log file should be on a dedicated disk. To minimize disk arm movements the blocks should be physically neighboured, as it is guaranteed by a file system based on extent-allocation. For recovery or undo operations there should be a fast access to the log records on the log file. Since the possibility of a disk failure is less a magnitude than that of single-node or system failures it is not necessary to administrate mirrored log files here.

Because disk space often is limited and there is no real need to store all dataunits which are logged until the end of the program, a log file design similar to [9] can be used. The log is written to a 'ring' of log files where a file which is no longer necessary for recovery – it does not include a most recent version of a dataunit – is used again. The problem with this optimization is that only the dataunits can be freed. The result parameters of actions are needed for the recovery of the Pact Program. This makes two different types of log files unavoidable, which results in two disk-writes for each commit and increased commit delay.

Note that in opposite to usual transaction processing system designs the action's updates are logged in Pact only during commit. This helps making logging faster, and therefore increases the throughput of the system and reduce the commit delay, because for each action there is only one disk-write for logging required.

Since in the Pact prototype there is a centralized Log Manager, it is easy to integrate a distributed Log Manager if there are some performance needs on massively parallel computers. The distributed version of the Log Manager is so easy to integrate, because actions execute on only one node. There are only marginal changes needed to the protocols described in this paper. The design is illustrated in Fig. 4. For best load-balancing each server node has a related Log Manager where it sends all its log messages. If the Supervisor has to recover dataunits it has to ask all Log Managers for their most recent version of the dataunit and choose the dataunit with the most recent timestamp.

With this Log Manager design and the commit protocol described in the section above it is possible to talk about the recovery of failures.

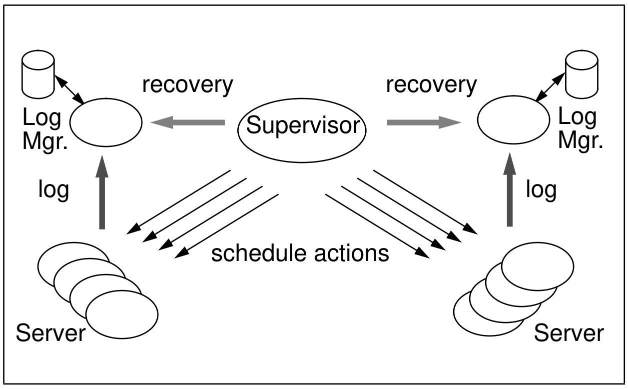

Fig.4. Distributed Log Manager

5.3 Recovery

While talking about recovery it is necessary to distinguish between different cases of failures. One category are single-node failures, the other are system failures – the crash of the complete system.

Remember the runtime system design, where Pact's system processes, i.e. Supervisor and Log Manager reside on different nodes of the system. This is primarily done to reduce the risk of getting a bottleneck in the system, but it is also useful for recovery of single node failures.

Therefore single-node failures can be distinguished between the failure of the Log Manager, the Supervisor, one of the server nodes, and the node where the Pact Program runs. In all these cases failures of nodes are recognized when an attempt to send a message to the failed node gives a 'node failed' error back. This requires that the underlying message-passing system has the ability to identify the failure of nodes. This is usually guaranteed for parallel computer systems but not for distributed systems. Whenever a process receives a 'node failed' error message, it informs the Supervisor which node fails in order to allow the Supervisor to start recovery.

Recovery of a server node failure:

1. The Supervisor sends a message to all server nodes left in order to inform them of the failed node.
2. The Supervisor and all server nodes close their connections to the failed node. The Node Managers send the identifiers of all dataunits they own to the Supervisor.
3. Based on this information the Supervisor updates his ownership table and determines the lost dataunits. The ownership attributes of lost dataunits are distributed among the rested Node Managers. Then the Supervisor sends the ownership table to all Node Managers.
4. After receiving this ownership table the runtime system resumes its work. Whenever a server requests a dataunit which has been lost, the Node Manager which owns it, gets it from the Log Manager.

During step 2. and 3. no server requests can be handled and therefore it is necessary to interrupt the complete runtime system. This is avoided if a different virtual shared data algorithm is used, where actual informations about the ownership of dataunits are available. This is a topic of further research. Although this algorithm is not optimal because of the reason described before, it needs for N server nodes only 3*N messages.

This is the most often used type of single-node recovery, because most of the nodes in the system are server nodes. A failure of the Log Manager is less probable. Now, let's have a look at the recovery of the Log Manager.

Recovery of the Log Manager:

1. The Supervisor starts the Log Manager on another node with the same log files again.
2. The Supervisor sends a message to all Node Managers which tells them that the Log Manager had been failed. After receiving this message the Node Managers again send log messages which have not been acknowledged yet. Multiple log entries are avoided by sending the action's action-identifier with each log message which identifies the action uniquely in the program.

This runs in parallel to the execution of actions, except that during recovery no action can commit. The number of messages needed for recovery is about N and the time the Log Manager is unavailable is about the time needed for process-creation.

The log manager recovery runs as independent as the recovery of the Pact (main) program.

Recovery of the Pact Program:

1. After the Supervisor recognizes the failure of the Pact Program it is started again.
2. The Pact Program sends its action calls to the Supervisor as it is done during normal start-up.
3. The Supervisor does not execute the actions again, but gets the results from the Log Manager and sends them back to the Pact Program, until it has reached the state it failed before.

During this recovery, normal execution can go on, because the Supervisor sends action-calls to servers, which are included in an execution plan it gets before the Pact Program crashes.

The fault of the Supervisor is handled like a fault of the complete system, because it has connections to all processes of the runtime system and therefore it is easier to restart it again. Hence, refer to the system recovery paragraph.

System recovery is done either by an explicit restart of the program by the user – this does *not* mean start the program again from the beginning, but begin restart-recovery – or by a system process started in the start-up script of the operating system. This system looks for 'open' log files and restarts the program.

Recovery of a system failure:

1. First the Supervisor is started again. It starts the Pact Program, all servers, Node Managers and the Log Manager.
2. After system start-up the Supervisor computes the ownership information and distributes it to the Node Managers.
3. The Pact Program sends action-calls to the Supervisor which receives the results from the Log Manager. If the Supervisor receives an action-call which is not executed before the crash, it is scheduled again.
   At the same time the Node Managers receive the last actual version of the dataunits owned by them from the Log Manager.

   Now the system is running again.

6 Conclusions and Further Work

Since fault tolerance is provided here in a fully user transparent way, the efficiency of this approach depends primarily on the correct choice of the action's granularity, because the action is the unit of recovery and parallelism. If the action's granularity does not fit with the underlying hardware and software the parallel program cannot run efficiently. Since this applies to parallel computing in general, it is especially important for this approach, because each action's commit needs some messages and a disk I/O. Although commit is done asynchronously to other actions' execution and therefore reduces the throughput only marginally, it increases the action's execution time. On the other hand, the actions should not be too coarse grained, because then there is too much work to be redone if a node or the complete system fails. But this is also true for all checkpointing schemes, where the distance between two checkpoints must be selected correctly by the programmer.

Now suppose the granularity is chosen well, how can one compare the Pact approach to checkpointing techniques? First, let us compare a local checkpoint with an action's commit. Suppose the work between two checkpoints is the same as within an action. Then an action just has to log its updated dataunits and some small amount of data – its result parameters. For writing a checkpoint there is much more work necessary, because for a checkpoint the complete state of all variables in the process is needed. This produces much more log data and longer delay.

Besides this, between two checkpoints there is extra logging needed for every message transfer – synchronous for pessimistic schemes, asynchronous for optimistic schemes – to avoid the global rollback to a consistent global state (the domino effect). Because Pact has integrated concurrency control on shared data this is not necessary for this approach.

For efficiency of recovery Pact lies in between the pessimistic and the optimistic checkpointing schemes. Pact's recovery of single-node failures is not as fast as in pessimistic schemes. In these schemes each process can recover independently from its checkpoint and subsequent logged messages. With the distributed virtual shared memory algorithm in Pact it is necessary to build a globally consistent ownership table, which needs O(N) messages for N server nodes. But Pact's recovery is much faster than optimistic checkpointing schemes, which need more than $O(N^2)$ messages [11].

Compared to the approach from Wu and Fuchs in [21], where each dirty page of a shared virtual memory is checkpointed, Pact's recovery algorithms need more messages and cannot provide the completely independent recovery of processes. But this advantage is purchased in [21] by using a centralized shared virtual memory manager, which does not fit with massively parallel computing. If they also use a distributed manager algorithm their recovery scheme is comparable to Pact's. Since their checkpointing scheme needs a disk-write for each dirty-page read, Pact only needs one disk-write for each action's commit, because of its log manager design.

But still, there are open research topics in Pact. The computation model is restricted to flat actions. For parallel programming nested actions – an action can call other parallel actions – are desirable. This decision is made, although there is a complete theory on nested transactions from Moss [17], but all algorithms are much more complex for nested transactions, and so the runtime overhead is much higher, too.

Another open research topic is a concurrency control mechanism with weaker – user defined – consistency, like described in the iterative solver example in section 5. This should provide more parallelism for parallel programs with many access conflicts on shared data.

Since the Pact prototype is currently implemented on a Tandem multi-processor system with moderate parallelism, the next version of the prototype will be implemented on a massively parallel Intel Paragon™ multi-processor system.

7 References

[1] Mike Accetta et al. Mach: A New Kernel Foundation for UNIX Development. In *USENIX 1986 Technical Conference*, pages 93–112, June 1986.

[2] J. Bartlett. A NonStop Kernel. In *Proceedings of the Eighth Symposium on Operating System Principles*, December 1981.

[3] A. Borg, J. Baumbach, and S. Glazer. A Message Passing System Supporting Fault-Tolerance. In *Proceedings of the ACM SIGOPS Symposium on Operating System Principles*, pages 90-99, Atlanta, October 1983.

[4] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle. Fault Tolerance Under UNIX. *ACM Transactions on Computer Systems*, 7(1):1-24, February 1989.

[5] Flaviu Cristian. Basic Concepts and Issues in Fault-Tolerant Distributed Systems. In A. Kershmer and J. Nehmer, editors, *Operating Systems of the 90s and Beyond*, pages 119–149, Dagstuhl Castle, Germany, July 1991. Springer-Verlag.

[6] Flaviu Cristian and Farnam Jahanian. A Timestamp-Based Checkpointing Protocol for Long-Lived Distributed Computations. In *Proceedings of the Tenth Symposium on Reliable Distributed Systems*, pages 12–20, Pisa, Italy, September 1991. IEEE Computer Society Press.

[7] Jim Gray. Notes on Database Operating Systems. In G. Goos and J. Hartmanis, editors, *Lecture Notes in Computer Science*, pages 393–481. Springer Verlag, 1978.

[8] Jim Gray. Why do computers stop and what can be done about it? In *Proceedings of the Fifth Symposium of Reliability in Distributed Software and Database Systems. Los Angeles*, January 1986.

[9] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, 1992.

[10] David B. Johnson and Willy Zwaenepoel. Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing. In *Seventh ACM Symposium on Principles of Distributed Computing*, August 1988.

[11] T. Juang and S. Venkatesan. Efficient Algorithms for Crash Recovery in Distributed Systems. In: *Proceedings of the Tenth Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 349-361, 1990.

[12] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[13] Kai Li, Jeffrey F. Naughton, and James S. Plank. Checkpointing Multicomputer Applications. In *Proceedings of the Tenth Symposium on Reliable Distributed Systems*, pages 2–11, Pisa, Italy, September 1991. IEEE Computer Society Press.

[14] Barbara Liskov. Distributed Programming in Argus. *Communications of the ACM*, 31(3):300–312, March 1988.

[15] Joachim Maier. Pact: A Fault-Tolerant Parallel Programming Environment with Virtual Shared Data. Technical Report, Stuttgart University, 1992.

[16] Joachim Maier. Pact – A Fault-Tolerant Parallel Programming Environment. To be published in: *Proceedings of the First International Workshop 'Software for Multiprocessors and Supercomputers: Theory, Practice, Experience'*, St. Petersburg, Russia, February 1993.

[17] J. Eliot B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. PhD thesis, Massachusetts Institute of Technology, April 1981.

[18] Michael L. Powell and David L. Presotto. Publishing: A Reliable Broadcast Communication Mechanism. In *Proceedings of the ACM SIGOPS Symposium on Operating System Principles*, pages 100-109, October 1983.

[19] Robert E. Strom and Shaula Yemini. Optimistic Recovery in Distributed Systems. *ACM Transactions on Computer Systems*, pages 204-226, August 1985.

[20] C.T. Wilkes and R.J. LeBlanc. Rationale for the Design of Aeolus: A system programming language for an action/object system. In *International Conference on Computer Languages*, pages 107–122, New York, 1986. IEEE.

[21] K. L. Wu and W. K. Fuchs. Recoverable Distributed Shared Virtual Memory. *IEEE Transactions on Computers*, 39(4):460–469, April 1990.