Supporting Action Management in Heterogeneous Distributed Systems

Edgar Nett

German National Research Center for Computer Science (GMD)
Postfach 1240
D-5205 St. Augustin 1, Fed. Rep. Germany
e-mail: nett@gmdzi.gmd.de

Abstract

An action scheme is proposed that reflects the cooperation with other autonomcus database systems and other models of data repositories like file systems or object stores. It is provided by the operating system level to support this kind of heterogeneity. Its key feature is that the provided mechanisms for recoverability are designed in an orthogonal manner meaning that it is not presupposed to conform to a particular syntactical concurrrency control criterion.

1. Introduction

Transaction management of today's database systems fails to comprise the cooperation with other autonomous database systems and other models of data repositories like file systems or object stores. Also, support provided by some operating systems so far does not reflect this kind of heterogeneity. To do so, the functionality provided by the operating system must be flexible enough to allow the other systems to additionally exploit available semantical knowledge regarding concurrency control as well as recovery. This requires that properties like isolation, atomicity, and durability can be unbundled and relaxed according to specific needs rather than being hard-wired in the transaction paradigm.

An action scheme is proposed that meets these requirements. Its key feature is that the provided mechanisms for recoverability are designed in an orthogonal manner meaning that it is not presupposed to conform to a particular syntactical concurrrency control criterion, e.g. serializability, or even to a specific implementation method, e.g. strict locking. The provided damage assessment is decoupled from any kind of recovery action to be done subsequently. The Damage Assessment Protocol (DAP) only determines the set of all affected data items that. They would be invalidated if automatic (syntactical) backward error recovery were applied. If so, a subsequent restoration of the recovery points identified as a result of the DAP would be locally done under the control of the respective data repository managers. This provides a lot more flexibility since applying syntactical backward error recovery is only an option provided by the operating system. This gives rise to a semantical evaluation of the damage performed by the affected systems themselves. According to the result, this may allow to proceed differently.

2. The recovery model

Without loss of generality we can assume that the distributed system clearly separates objects which carry state from computations which carry on system activity by changing the state of objects via the execution of some operations. Thus, only objects are the target of any recovery action and we are able to describe our recovery model independent from a specific computation and data model, respectively.

During its lifetime the actual state of an object is a candidate to serve as an input parameter for a final set of operations O. We assume a read/write semantics, i.e. the elements of O only differ with respect to whether they produce a new state or preserve the object's state. Thus, from its creation until its actual state x^n, every object x has gone through a sequence of states $(x_o{}^1, \dots, x_{o'}{}^n)$ where o and o' denote two operations from O that have produced the respective states. We only specify the operation responsible for an object state if necessary.

An object state that is restorable is termed a **recovery point** of the respective object. A recovery point is established by performing a *save* operation. Only actual object states can be saved. By performing the restore operation for a recovery point it becomes the new actual state of the object whereas all subsequent states become invalid. When a recovery point is *discarded* it looses its ability to become restored. A subsequence $(x^s, \dots, x^n)$ $(1 \le s < n)$ may contain several recovery points. For each object state x^i $(s \le i \le n)$, x^{irec} denotes the corresponding recovery point that will be restored when x^i is invalidated. The most recently established recovery point of an object x is said to be *active*.

Object states may depend on each other in the following way: Let x^i and y^j represent two states of possibly different objects x and y and let o denote the operation that has produced y^j. Then, x^i-->y^j meaning that y^i depends on x^i holds if o has used x^i as an input parameter in order to deliver y^i.

The relation --> is termed **object dependency**. Backward error recovery assumes that the relation --> implies a total order on the sequence of states of every object x such that x^i-->x^{i+1} holds for $1 \leq i \leq n-1$ where n is the actual state. (This assumption does not mark any loss of generality because a new object state that is not dependent from its predecessor - in the literature also termed "blind write" - can be modeled as the initial state of a new object.)

Now let us assume that some error detection has declared an object state x^i to be erroneous. Obviously, errors do propagate by means of the object dependency relation. Thus, all object states that are dependent on an erroneous state are erroneous themselves from the very beginning of their existence. They can be described by x^{irec}-->* which denotes the transitive closure of the object dependency relation emanating from the corresponding recovery point of x^i. We term this effect **forward error propagation.** Let $Z := \{z_1=x, \dots, z_m\}$ denote the set of all objects that are affected by the forward error propagation and let z_j^i ($2 \leq j \leq m$) denote those object states that have imported the error from another object by being dependent on an erroneous state of that object. Then, all the respective recovery points z_j^{irec} have to be restored leading to an invalidation of additional object states. This effect is termed **backward error propagation.** Obviously, this may lead to another round of forward error propagation and so forth. The process of finding the affected objects and determining the recovery points that eventually have to be restored is termed **damage assessment.**

The **recovery region** $R(x^i)$ of an object state x^i is defined by the set of all object states that have to be invalidated according to the damage assesssment when x^i is declared to be erroneous. Let X^i be a set of erroneous object states. Then, $R(X^i)$ represents the union of the recovery regions of the elements from X^i. The **periphery** $R_p(x^i)$ of a recovery region $R(x^i)$ is marked by the set of recovery points that have to be restored as a result of the subsequent recovery action. An object state which no longer can be an element of any recovery region nor its periphery is said to be **safe.** A safe recovery point will never be restored or invalidated during some recovery action and, therefore, can be discarded.

Let x^i be the youngest state of an object x that is safe, i.e. for any other safe object state x^j it must hold that j<i. Then, x^{i+1}, which represents the immediate successor of x^i in the sequence of states of x, is the only state of x that cannot be an element of any recovery region but, still, can be an element of the periphery of some recovery region. Obviously, x^{i+1} cannot be an element of any recovery region since otherwise x^i cannot be safe. Also, all object states older than x^{i+1} are safe. Therefore, let us assume that there exists an object state x^j which is younger than x^{i+1} and which has the same property like x^{i+1}. Then, however, x^{i+1} would also be safe which contradicts the assumption that x^i was the youngest one with that property.

The in such a way distinguished object state x^{i+1} is called the **commit point** of x. During its lifetime x can have at most one commit point at a time. The set of all commit points is said to be the **commit line** of the system. Obviously, in order to be a candidate for commitment, an object state must satisfy the condition that it is no element of the transitive closure of the object dependency relation of any object state in the system which is neither a commit point or also a candidate for becoming a commit point nor safe.

In case that system progress is manifested through individual computations observing the atomicity property, the following two conditions must be satisfied:

- every computation that performs a write operation on an object x for the first time must ensure that the object state of x, say x^j, which serves as an input parameter for that write operation is established as a recovery point. Analogously, x^j is termed the before_image of the computation with respect to object x. The set of all object states that represent before_images for a computation C is said to be the **recovery line** RL_C of C.
- For each object state x^i which is accesssed by a computation C it must hold: RL_C is a subset of the set that results from the union of $R(x^i)$ U $R_p(x^i)$.

To satisfy the second condition it becomes necessary to describe more precisely the process of damage assesssment. This relates to the procedure of backward error propagation where we now have to specify exactly the state x^{irec} which denotes the recovery point that will be restored if the object state x^i is erroneous. The recovery point x^{irec} is defined by the before_image of that computation which has been responsible for the creation of the erroneous object state x^i. Furthermore, it must be ensured that, if one element of a recovery line RL_C is restored, RL_C as a whole must be restored. To accomplish this, the **recovery point dependency** relation ==> has to be introduced:

For two recovery points x^i and y^j it holds that x^i==>y^j meaning that y^i is recovery dependent of x^i iff y^i has to be restored whenever x^j is restored.

It is required that the relation ==> forms a total ordering on a a recovery line RL_C, i.e. all pairs of elements of RL_C are recovery dependent. In the normal case, this relation is bidirectional. This only changes when nested computations have to be considered. Let C' be nested within C, x^i a before_image of C and y^j a before_imageof both C and C'. Then, it only holds that x^i==>y^j and not vice versa.

Let x^i_C denote an object state that is produced by computation C. If x^i_C is detected to be erroneous then, the element of RL_C that represents the before_image od C concerning x is restored as a result of error propagation. Consequently, all elements of RL_C will be restored because of the recovery point dependency relation. This guarantees that all elements of the recovery line are contained either in the recovery region of x^i_C or in its periphery even if there is an object y that is represented in the recovery line of C but not in the transitive closure x^i_C-->*. Hence, the atomicity property is observed if C fails.

Let A_C denote the set of all after_images of a computation C. C is said to be committed if all elements of A_C have become commit points. The atomic execution of computations designates them for being the smallest units of the computational activity of a system and, therefore, also for its progress. This implies that the forward move of the commit line of the system occurs such that at least all objects that are involved in the commitment of one computation C have to update their commit point. Regarding the potential loss of computational activity it follows that the periphery of each recovery region can be described by the union of the recovery lines of a final number of computations. As a consequence, a necessary condition for the domino effect is the existence of a computation with a recovery line that is not a subset of the commit line. Otherwise, the commit line cannot be moved forward anyway because of the atomicity property of computations. From that it can be concluded that no revocation of a computation can lead to a domino effect if for each object state x^i and for every computation C it holds that RL_C is part of $R_P(x^i_C)$. By definition, this is always true at the very beginning of a computation. Then, if no damage assessment procedure will involve the backward expansion of a recovery region, this premise will always hold. The expansion of a recovery region is termed backward, if it implies the substitution of recovery points in the respective periphery by older recovery points of the same objects. Pictorially spoken, the periphery has to be moved backward. Otherwise, if the periphery is only enlarged or remains unchanged, the expansion is termed forward. There are different approaches to cope with the phenomenon of backward expansion giving the user the flexibility to select the most appropriate one for the respective application [1].

3. Related work

So far, in the literature there can be found two principal approaches for backward error recovery in distributed systems. One does not impose any a priori conditions on, according to our model, recovery regions and their respective periphery that have to be observed by the system and, therefore, is called unplanned [2,3]. This means that no constraints are imposed on the shape of any periphery $R_P(x^i_C)$ of a computation C. This approach represents the one extreme within the spectrum of conceivable proposals where ease of recovery and the care for minimizing the damage in case of errors are sacrificed for unrestricted communication and the speed of the actual computational progress during normal processing. Therefore, to go along with this maxim sometimes is also termed optimistic [4,5]. This characterization immediately leads to a major drawback which is that it cannot prevent the domino effect. On the one hand, optimistic recovery does not need the support of further mechanisms like those conceived for controlling concurrency. On the other hand, this renunciation of any "external" support is not compensated by "internal" measures that prevent the possible occurrence of the domino effect.

Another big disadvantage is that it does not consider atomic computations, i.e. the consideration of what we designate as recovery dependencies which do exist independently. As a consequence, if recovery has to observe the failure atomicity, either optimistic recovery cannot be applied or it has to be completed by additional measures. Taking the latter option, however, would violate its purpose of being applicable independently of the existence of other system features and being transparent to the application interface.

An interesting alternative is proposed in [6]. It presents a scheme for the execution of concurrent processes where the processes are allowed to send to each other information that has not been completely validated and hence, may be revoked later. The processes are assumed to be heavy weighted like UNIX processes and are subdivided in recovery regions. It is not clear whether the proposed approach is independent from this computational model. It is shown that during recovery two consecutive recovery regions of a process will never have to be invalidated. This is guaranteed by allowing the process to leave a recovery region only when certain conditions are met. However, it is not clear whether these conditions always will hold eventually.

The other main approach pursues just the opposite extreme. It is termed planned because it requires that the entire periphery must always be known prior to the creation of the respective object state. More precisely, for every computation C it must hold that $R_P(x^i_C) = RL_C$ for all object states x^i which are created by C. This is normally achieved as in the transaction concept by relying upon concurrency control methods that have to enforce the most restrictive consistency constraint which is the isolation property. In addition, all elements of RL_C have to be commit points. As a consequence, damage assessment can be totally omitted which again implies that their is no need for recording and subsequently evaluating the arising object dependencies. Also, the domino effect is prevented. Obviously, in this approach the total concern is focussed on exceptional processing in which case ease of recovery and minimal invalidation of computational activity is of predominant interest. Hence, loss of efficiency and processing speed by exploiting the communication and concurrency inherent in the concept of distributed processing which even becomes more severe the more reliable the system works. Therefore, to squeeze backward error recovery in such a tight conceptual corset is also named as pessimistic.

Based on the recovery model just described we are able to propose dynamic actions as the abstraction which is provided by the system at the interface to the application. This means that dynamic actions can serve as the atomic computational unit on which the client can structure his application.

The fundamental difference between the transaction paradigm and dynamic actions is that the former relies on the isolation principle whereas the latter supports cooperation. This becomes the more predominant when the cooperating entities are getting larger and more self-determined. In this view, the system concept as depicted in section 5 can be described as the cooperation of multiple subsystems termed resource managers based on dynamic actions. It is often emphasized that, in order to integrate a database management system it must still be possible to exploit database - specific semantics. The functionality provided by the operating system should be flexible enough to allow the use of such semantic knowledge, e.g. with respect to concurrency control and high-level recovery like the use of compensating actions. However, this requires that, as it is possible in the proposed dynamic action scheme, properties like isolation, atomicity, and durability can be unbundled and relaxed according to specific needs rather than being hard-wired in the transaction paradigm.

4. Dynamic actions

From the user's point of view it makes sense to distinguish three different states a dynamic action may run through during its lifetime. The begin of its lifetime cycle is obviously marked by the event of its *creation*. From now on it is in the **active** state. It remains active as long as there are some operations not finished or still to be executed, resp. When this condition ceases to exist, which represents the end of the active state of the dynamic action, it enters the state **terminated**. It holds this state as long as at least one of its after_images is still not committable. After, its state changes to **completed** meaning that from now on all after_images are committable. It follows that it no longer can be affected by a computational fault, since no error propagation would be able to reach its before_images. There are two possible events that mark the end of an action's lifetime. The one represents the 'all' effect, i.e. its success, which occurrs when all after_images have been committed. Therefore, this event is also termed the *commit* of the dynamic action. This event can only occur during the state of being completed. From now on also no node fault can affect its results. The other possible ending event, termed *abort*, signals its failing and represents the beginning of the error processing phase that will lead to the nullification of all its effects. In contrast to the commit, the abort can occur at any time during the lifetime of a dynamic action. Note that, in principle, the actual state of a dynamic action can be disclosed whenever one of the object states it has created is wanted to be accessed from another action. However, to proceed like that in some cases may be overruled by the concurrency control.

It is emphasized that preserving the atomicity property is orthogonal to the way the computational activity proceeds in the system. This is possible since the applied recovery model allows the dynamic growth and shrinkage of individual recovery regions independent of the computational structure of the system. This reflects one aspect of the 'dynamics' that characterizes the proposed scheme. Hence, dynamic actions impose no restrictions on the various consistency policies that different users may adopt to be observed by their application. Actually, it should also contain a variety of mechanisms for concurrency control in order to be able to meet the requirements of the policy selected by the application. The user still may have the freedom to care for an adequate consistency within his application itself and still rely upon the atomicity property of dynamic actions. From that we can identify a structure consisting of three different layers that are independent from each other. They are depicted in Fig. 4.1.

The first layer comprises the mechanisms for the general management of dynamic actions. Its main task is to organize the action management in a distributed fashion where all nodes of the system that are involved in a dynamic action have to cooperate. In addition, it has to support the nesting of actions. In [7], the details are described for the management of distributed, nested dynamic actions. Because they also apply to the management of dynamic actions, they are not discussed more thoroughly. We only want to point out an additional aspect with respect to the nesting facilities.

It is now possible or even necessary that a nested structure can be initiated not only from the application level but also by the system itself, e.g. in order to prevent the danger of an arising domino effect or in order to control the growth of recovery regions. Both examples refer to the needs of the third layer. In transactional systems, nesting is only viewed as a means with respect to the second layer, i.e. for concurrency control. There, it is used to control concurrency within a single transaction according to the same consistency notion that is applied between different transactions. Of course, this traditional aspect of nesting also can be applied in the context of dynamic actions following the consistency policy as selected by the application. Hence, the action management has to distinguish between these two aspects of nesting.

In other words, this means that also nested actions may be designed having different properties. They may observe different consistency criteria. They may be part of the same recovery region or they belong to different ones. We can even think of nested actions having the permanence property. This would imply that the top-level action would not be atomic. Modelling interacting services in a telecommunications environment would be an application for such a case. There, an important requirement is that those services or features which represent the independent units of work have to cooperate whenever necessary to establish and control communications between parties. Thereby, new services are permitted to be added as well as to be cancelled dynamically [8,9]. This can be seen as a new, more comprehensive unit of work (service) which, however, should not have the atomicity property.

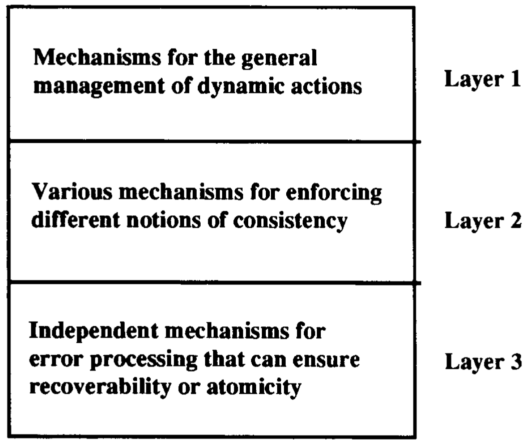
Mechanisms for the general management of dynamic actions
Layer 1
Various mechanisms for enforcing different notions of consistency
Layer 2
Independent mechanisms for error processing that can ensure recoverability or atomicity
Layer 3

Fig. 4.1: Layered structure

The second layer contains various mechanisms to enforce the consistency policy selected by a distinct application starting from no restriction at all until complete isolation of the individual computations as it is required for transactions. Note that we are now able to distinguish between the requirement for isolation and serializability meaning that to observe the serializability criterion does no longer necessitates the same severe restrictions to be imposed on normal processing. In case that also serializability is not required or even not acceptable with respect to the application behind, only the various mechanisms to avoid backward error propagation are offered.

The third layer refers to the mechanisms for error processing. In order to be general purpose and to apply also to heterogenous environments it has been designed to be always able to care for recoverability and atomicity independent of what is done in the second layer. The main mechanisms needed are [10,11]:

- a damage assessment graph which represents a distributed data structure where each local graph reflects the computational activity and the dependencies of the respective node.
- a damage assessment protocol to determine the abort set which constitutes the set of all affected object states.
- a commit protocol to move forward the commit line of the system.
- an atomic broadcast protocol.

The resulting complex task of this layer should be made completely transparent to the application level. This has also to include efficiency considerations. Therefore, it should also be possible to optimize the combined use of the mechanisms provided by the different layers by adapting the general recovery model dynamically to the specific constraints imposed by a given application when applied in the respective dynamic actions. This represents another aspect of 'dynamics' inherent in the proposed abstraction. Imagine, e.g., serializability is required and it has been decided to meet it by using non-strict 2-phase-locking [12,13]. Then, the corresponding damage assessment graph must not be checked for the possibility of backward expansion in order to prevent the domino effect. If, to put it to an extreme, isolation has to be maintained as in conventional transactions, not a single object dependency but only recovery point dependencies need to be recorded.

5. System's view

As already pointed out earlier dynamic actions represent a computational abstraction provided by the (operating) system to support the user in designing distributed applications. For the system itself we take the same view as most of the modern operating system do, e.g. Mach, OSF, Chorus, Amoeba. In principle, it is structured in two layers where the lower one, termed micro or low level kernel, respectively, contains the common basis for all the services provided by the upper layer.

Low level broadcast/multicast protocols should also be executed on that level, either by integrating them into the kernel or by providing an interface that permits to plug them in, a technique, e.g. used in the BirliX operating system [14]. On top of this kernel the various services, sometimes also called managers, are realized that should be provided by a distributed operating system.

Regarding the architectural integration of dynamic actions as a manager we assume the same structure that is proposed in the X/Open model for distributed transaction processing and that is also applied for the distributed transaction facilities in the QuickSilver distributed operating system [15] as well as for the PROFEMO transaction mechanism which is developed further and integrated in such an architecture in the RelaX project [16]. It is sketched below in Fig. 5.1 for one node. Thus, all approaches can profit from being clearly separated from the management of different ressources representing different models of persistent data that may be accessed (e.g. files, data bases, abstract data types). It allows to realize an integrated action concept across the different resource types.

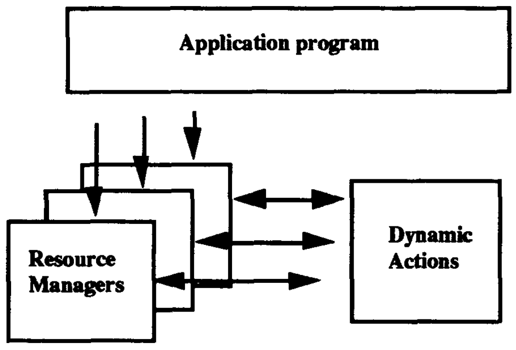

Fig. 5.1: System Integration

To make distributed computations fault tolerant their operations that may have to be performed on multiple resource managers are composed into dynamic actions. Therefore, the management layer of dynamic actions has to communicate with the respective resource managers to integrate their work. Examples of resource managers include file systems, database systems, object repositories. Each resource manager can be developed and administered independently and resource managers can be layered on other resource managers. For example, layering an object manager on top of all other resource managers that also represent long-living data provides and object-oriented view of the system. Independent development of resource managers means that the designers do not have to anticipate all of the ways in which applications will compose operations and, hence, supports heterogenity.

6. Impact on error handling

In our recovery model, errors propagate due to data dependencies meaning that data being produced as a result of an operation that has used other data as input parameter that later on is detected as being erroneous is also declared to be erroneous, i.e. to be part of the damage. But this does generally not imply that the caused damage inevitably leads to a failure. A failure relates to the semantical properties of what is an acceptable result as they may be described in the specification. Damage represents a syntactical property which does not consider any semantics. Identifying damaged with failed is an approach where the good semantical properties (no failure!) are strived for on the basis of the syntactical properties (damage is recovered!) that the system guarantees. Obviously, if damage should be transparent to the application interface, this is the only way to go. The same applies to serializability, which can be seen as a syntactical consistency criterion from which semantical consistency should be derived.

In addition to the ordinary methods of backward error recovery, independent damage assessment leaves different alternatives how to proceed in such a situation depending on whether the damage assessed does severely affect the expected result as well as on the support the particular application can contribute to recovery. It may turn out that the dependency which imported the damage did not affect at all the produced result. Imagine, e.g., that operations have been performed dependent on the positive value of a particular integer object which turnes out tobe erroneous but not the fact that it is positive. Then, these operations or even the whole action does not have to be aborted. Another reason for not performing any kind of at least backward error recovery is that due to an error an action has only succeeded partially. This outcome cannot be handled by concepts that only rely on the all-or-nothing property. Now, it might be considered to initiate a *corrective* action meaning that the shortcomings in the effect of the previous action are adjusted.

Furthermore, global actions may be hindered from initiating syntactical backward error recovery simply because a participating system refuses to accomplish its part of restoring the respective recovery points on behalf of its autonomy. One reason might be to avoid the cascading aborts of a bunch of depending local actions which could not be tolerated. In this case, initiating a corrective action which calls another system to take over would be a solution. Autonomy reasons may also imply that the nested local actions commit or abort irrespective of the outcome propagated by the manager of the respective global, top-level action. As a consequence, the only way to perform backward error recovery is to run a *compensating* action.

A compensating action is able to undo all the effects produced by that action that has to be reversed. Again, this leads to the restoration of the respective recovery points, but in contrast to syntactical backward errror recovery this is achieved by exploiting the semantics of the particular application. Hence, it can be viewed as performing semantical backward error recovery. There are many applications where running compensations is done not because it is the only way to go. Rather, it represents an attractive alternative to the syntactical variant because the compensating action is easy to derive and/or produces less overhead. Important to note that still the damage assessment is a necessary requirement. The information about wat has to be undone must be known in order to devise the corresponding compensating action(s).

7. Conclusion

An action scheme for heterogeneous distributed systems iss proposed. Thereby,heterogenity reflects the cooperation of multiple autonomously designed subsystems termed resource managers, which has been our primary design purpose, as well as the interoperability of multiple autonomous computer systems.

Dynamic actions are general purpose but still adaptable to the specific application such that substantial loss of performance due to the provision of unnecessary functionality is avoided. This is achieved since they do not prescribe any policy with respect to concurrency control and error recovery. Instead, they provide some kind of a toolbox wich allows the application programmer to define its own one. When starting a dynamic action, the programmer is offered a list of hierarchally structured parameters that allow him to select the consistency criterion to be observed including none. If desired, he can also vote for the mechanisms to realize it. Accordingly, the system automatically selects the adequate error processing policy. He may also influence, if there is any choice left by the used concurrency control mechanisms, they way his dynamic actions do proceed by determining the kind of objects that are allowed to be accessed.

8. References

[1]E. Nett. Supporting Fault-tolerant Computations in Distributed Systems, *Habilitationsschrift*, Bonn,1991

[2]T. Anderson and P.A. Lee. Fault Tolerance: Principles and Practice. *Prentice-Hall International, Inc*, 1981

[3]W. Wood. A Decentralized Recovery Control Protocol. *FTCS-11*, 1981

[4]R. Strom and S. Yemini. Optimistic Recovery in Distr. Systems. *ACM Trans. on Comp. 3 (3)*, Aug. 1985

[5]D. Johnson and W. Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. *Proc. 7. ACM Symp. on Principles on Distr. Comp.*, Aug. 1988

[6]K.H. Kim, J.H. You, and A. Abouelnaga. A Scheme for Coordinated Execution of Independently Designed Recoverable Distributed Processes. *Proc. 16. Int. Symp. on Fault Tolerant Computing Systems*, Vienna, July 1986

[7]R. Schumann. Transaktions-Verwaltung in einem verteilten objektorientierten System. *GMD-Studie No. 134*, Birlinghoven, 1988

[8]G. Gopal and N. D. Griffeth. Software Fault Tolerance in Telecommunications Systems. *4. ACM SIGOPS European Workshop*, Bologna, Sept. 1990

[9]F.Thomas. The feature interaction problem in telecommunications systems. *Proc. 7. Int. Conf. on Softw. Eng. for Telecommunications Switching Systems*, July 1989

[10]E. Nett and R. Schumann. Integrating Fault-Tolerance and Real-Time Requirements of Distributed Systems. *Proceedings 2. Workshop on the Future Trends of Distributed Computing Systems*, Kairo, September 1990

[11]R. Schumann, R. Kröger, M. Mock, and E. Nett. Recovery Management in the RelaX Distributed Transaction Layer. *8th Symp. on Reliable Distri. Systems*, 1989

[12]E. Nett, J. Kaiser, and R. Kröger. Providing Recoverability in a Transaction Oriented Distributed System. *Proc. of 6th Int: Conf. on Distributed Computing Systems*, Cambridge, Mass. 1986

[13]E. Nett, R. Kröger, and J. Kaiser. Implementing a General Error Recovery Mechanism in a Distributed Operating System. *FTCS 16*, Vienna, Austria,1986

[14]H. Härtig, et al. Distribution and Recovery in the BirliX Oper. System. *Informatik-Fachberichte 130*, Febr. 1987

[15]R. Haskin, et al. Recovery Management in Quicksilver. *ACM Trans. on Comp. Syst. 6(1)*, Febr. 1988

[16]R. Kröger, et al: RelaX-An Extensible Architecture Supporting Reliable Distributed Applications. *9th Symposium on Reliable Distributed Systems*, 1990