CORE
Metadata, citation and similar papers at core.ac.uk
Provided by KITopen

Fakultät für Informatik

76128 Karlsruhe

Flexible and Transparent Fault Tolerance for Distributed Object-Oriented Applications

März 1995

Dietmar A. Kottmann
Prof. Dr. Alexander B. Schill

Universität Karlsruhe
Institut für Telematik

Interner Bericht 20/95

This report describes an approach enabling automatic structural reconfigurations of distributed applications based on configuration management in order to compensate for node and network failures. The major goal of the approach is to maintain the relevant application functionality after failures automatically. This goal is achieved by a dedicated system model and by a decentralized reconfiguration algorithm based on it. The system model provides support for redundant application object storage and for application-level consistency based on distributed checkpoints. The reconfiguration algorithm detects failures, computes a compensating configuration, and realizes this new configuration. The report emphasizes flexibility in the sense of adaptable levels of fault tolerance, as well as transparency in the sense of fully-automatic reaction to failures.

Key words: Distributed applications, object-oriented systems, dynamic reconfiguration, fault tolerance, dependability, configuration management

The work presented in this report describes an integrated approach for providing fault–tolerance to distributed applications in a transparent manner. The basic idea is to exploit the capabilities of application–level *configuration management* [IEE92], the task of instantiating, allocating, and interconnecting application modules initially, as well as during dynamic configuration changes. The ability to perform such dynamic changes is the starting point for our approach, that combines reconfiguration with means for percepting node and network failures and for keeping application objects consistent. Concerning failures, we assume fail-stop behaviour of nodes and network components [SCS83]. In summary, the major contributions of the report are as follows:

- *Distributed and decentralized reconfiguration algorithm:* We present a new distributed algorithm performing automatic application reconfigurations after (node or network) failures. Due to its completely decentralized nature, the algorithm is also able to cover the problem of network partitions.
- *Mechanisms to guarantee consistency after reconfiguration:* An important property of the reconfiguration algorithm is that it guarantees consistency of ongoing computations. This is achieved by underlying system support for coordinated distributed checkpoints.
- *Support for flexible levels of fault tolerance:* Based on an underlying system model that allows for selected replication of application data, an application administrator can determine the desired level of fault tolerance explicitly (in terms of the set of potential failures to be tolerated).
- *Determination of replica placement based on the structure of an application:* As the configuration manager knows about interobject bindings, locations of replicas are selected in a way to ensure that those part of an object mesh, which are necessary to provide a certain functionality, are available at specific nodes with a high probability. This involves changing the locations of replicas in the case of a failure, eventually combined with the installation of new replicas.

As a foundation of our approach, we use a dedicated system model based on the *passive object model* described in [CHC91]. Objects (data structures encapsulated by a set of operations) are implemented by a distributed extension of C++; this supports location independent object invocations and a limited kind of dynamic object mobility similar to [JLH88]. A distributed application consists of communicating objects distributed to different nodes. Parallel and distributed object computation is initiated by concurrent lightweight threads, each invoking (potentially distributed) sequences of object operations. Of the general features of object orientation, data encapsulation is an important prerequisite for our approach, while inheritance is helpful for the internal implementation but is not a necessary part of our application model. Concerning distribution of object implementation code, we rely on the simple assumption that code for a given object type is replicated whereever an object of that type should be allocated. Dynamic code installation would enable a more flexible approach but is rather orthogonal to the concepts described in this report and could therefore afterwards be integrated.

For our approach, the passive object model has been augmented with the facility to write distributed object checkpoints during computation, and with a separate, explicit application configuration management facility according to [KMS89]. Details of these aspects are described later.

The report is organized as follows: Section 2 describes our approach from a conceptual point of view. First, the goals of transparency and flexibility of fault tolerance are clarified. Then our basic configuration management facilities are described by an example application, and the overall structure of our reconfiguration algorithm is presented. Section 3 discusses the technical details of our approach, namely the algorithm to map an application configuration to an underlying distributed system, the mechanisms to replicate application objects in order to enable reconfiguration after failures, and parts of the techniques to guarantee application consistency after reconfiguration. Finally, the reconfiguration algorithm is discussed in more detail based on these technical foundations. Section 4 presents initial experiences with the approach and section 5 compares our solution to approaches in several related areas. Section 6 concludes with an outlook to future work.

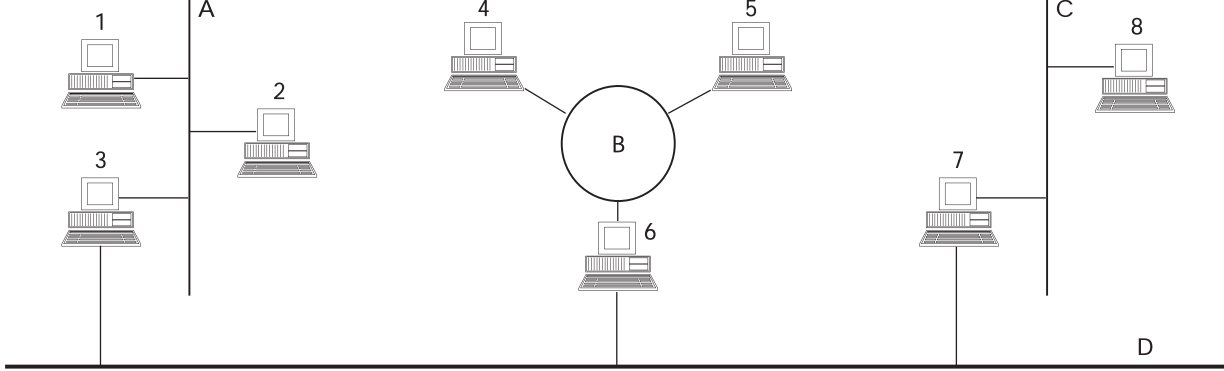

Figure 1: Distributed system example

2 Conceptual Approach

2.1 Major Goals

The general guideline of our work is the development of a scheme that provides flexible and transparent fault–tolerance to a distributed application.

Transparency means that the functionality is made available without taking explicitly care for dependability within the application code. This is achieved by two correlated means: First, our reconfiguration algorithm performs automatic failure detection, application configuration analysis, computation of a new configuration to compensate failures, and realization of the dynamic reconfiguration. Secondly, our system model enables reconfiguration based on location independent invocation, supports the required information redundancy by the facility for replicated object checkpointing, and guarantees application-level consistency by coordinating checkpoints globally.

Flexibility can be divided into three issues: compliance to the specific failure characteristics and intended usage of a given distributed system, selectable degree of availability for different parts of a distributed application, and flexible placement of redundant information for those parts. For example, the set of failures that have to be tolerated by an avionic system is broader than the corresponding one for an electronic mail system. This diversity should be managed in a manner that is independent from the application code. Therefore, a separate definition of the kind of failures to be tolerated, and of the availability requirements for different parts of the application is provided to achieve the stated compliance, whilst the flexible placement is computed automatically based on this information.

2.2 An Example

For the exemplification of the ideas our scheme is based on, we use the distributed system presented in figure 1, together with a simple application presented in figure 2.

The distributed system consists of a backbone (D) that connects three LANs (A, B, C). The LANs connect the nodes of two different general departments ({1, 2, 3}, {4, 5, 6}), and the nodes of the personnel department ({7, 8}).

The application consists of front–end objects for the departments (HD1F, HD2F), a front–end object for the personnel department (PF), and two storage objects (PD, S). The arcs in the figure depict potential call relationships between objects. The first storage object PD maintains all personnel records of the employees. The second storage object maintains the qualifications and rankings of the employees, and the sum of the employee wages for each department. Object PD has access to S for keeping the sums consistent to the wages of each employees in case a call to PD changes some wages. To keep the example

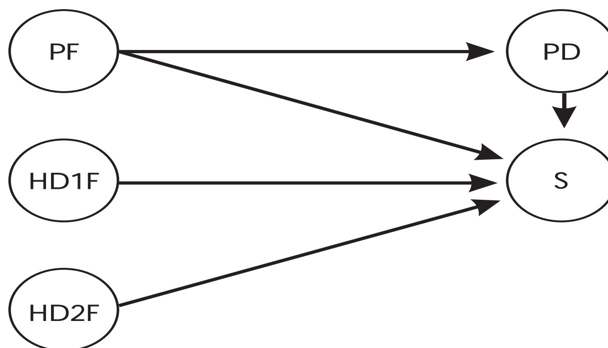

Figure 2: Example application

simple, there is just one object S instead of an object S_i for each department, which might be a better modelling of the application semantics.

An example of reconfiguration behaviour that can be achieved automatically with our approach in case of a failure is as follows: If the backbone fails, we want the system to react automatically in a way that the personnel department keeps unlimited access to all personnel-related data (stored in objects PD and S), while accepting access restrictions of the department front–end objects to read–only lookup operations on data stored in object S. This restriction keeps the managers of the general departments from updating the rankings, but is necessary to prevent the states of copies of S in different partitions from diverging. Later we will shortly discuss the possibility to use a storage object S_i for each department in order to allow the managers to update the rankings in case of a failure, too.

2.3 Configuration Management

To provide the required flexibility of application management, explicit control of the structure of a distributed application is needed. This is provided by a configuration management facilitiy along the lines of the *Conic* system [KMS89]. Our approach consists of a declarative configuration language to specify an application and system configuration, and of configuration managers mapping the specification to the underlying distributed object-based environment. In general, an application entity at the configuration level is mapped to a C++ object of a specific class that is a subclass of a dedicated configuration management class.

In the following, the configuration language is illustrated by specifying parts of the example application of figure 2 in a simplified way; after specifying the relevant object types, it is shown how the object instances of these types are declared and interconnected via *logical links*. These links are used at runtime for indirect addressing of target objects within remote C++ invocation requests. They originate from *reference variables* declared for each object type if required (e.g. for the personnel department); fields and arrays of reference variables are also supported as illustrated below. Hence reference variables are a means for controlling remote access just as the well known concept of *smart pointers* is a means for controlling memory usage and allocation. The logical interobject link structure is independent from the physical node interconnection structure; the latter one is specified by a dedicated topology graph. Moreover, object operations are declared, too, in order to map them to operations within the C++ extension. For simplification, some of the parameter types are omitted; object placement is also not shown here (for details see section 3.1 and [SCZ93]).

```
OBJECT_TYPE StorageSystemType;          // Storage of different records
  REFERENCES otherStore: StorageSystemType;
                                        // Optional ref. to other store
  OPERATIONS store (RecordType,string); // Operation to store a record
             retrieve (RecordType,string);
```

```
END_OBJECT_TYPE

OBJECT_TYPE DepartmentType;              // General department
  REFERENCES storage[*]: StorageSystemType;
                                         // Reference for logical link
  OPERATIONS manageProjectData (...);    // Edit project information
             performRanking (...);       // Ranking tool support
             // other operations
END_OBJECT_TYPE

OBJECT_TYPE PersonnelDeptType;           // Personnel department
  REFERENCES storage[*]: StorageSystemType;
                                         // Reference for logical link
  OPERATIONS modifyPersRecord (...);     // Access and operate on rec.
             addNewEmployee (...);       // Add a new employee
             // other operations
END_OBJECT_TYPE

OBJECTS HD1F, HD2F: DepartmentType;      // General departments
PF: PersonnelDeptType;           // Personnel department
PD, S: StorageSystemType;        // Storage systems
END_OBJECTS

OBJECT_LINKS HD1F.storage[1] WITH S;     // Dept. 1 -> Storage S
     HD2F.storage[1] WITH S;     // Dept. 2 -> Storage S
     PF.storage[1] WITH PD;      // Pers. Dept. -> Storage PD
     PF.storage[2] WITH S;       // Pers. Dept. -> Storage S
     PD.otherStore WITH S;       // Storage PD -> Storage S
END_OBJECT_LINKS
```

A compiler for the configuration language has been implemented using *LEX* and *YACC* (see section 4). The backend generates an internal configuration representation that is loaded by the initial configuration manager in order to set up the initial configuration.

Dynamic configuration changes can also be requested in a similar notation shown by a small example to introduce a new department:

```
CREATE HD3F: DepartmentType;
LINK HD3F.storage[1] WITH S;
```

These change requests are given to the dynamic change manager that executes them on the real configuration. However, in the following kernel part of this report, we only rely on a similar procedural interface to request dynamic reconfiguration internally and do not focus on changes based on interactive requests.

For the specification of the physical structure, a similar language is provided. Physical configuration changes, e.g. to include new nodes, can also be requested.

2.4 Dynamic Reconfiguration in the Case of Failures

Dynamic reconfiguration is performed if a failure is detected that obstructs the normal communication based operations. Application activity is suspended for affected objects, a new configuration is established, and the activity is resumed.

The reconfiguration may include the allocation of copies of objects that were once placed on a failed node or a node that isn't reachable any more. For this allocation after failure, the internal state of objects has to be replicated. If a failure leads to a partitioning, objects may be allocated in all resulting partitions if the necessary state information is accessible. Consequently, techniques are required for achieving consistency between objects surviving a failure on one hand, and newly allocated ones on

access to duplicates would in general prevent two configurations from merging automatically into a new, consistent one when two partitions are united later after recovery of a failed component.

These considerations lead us to the following algorithm for dynamic reconfiguration:

1. *Fault detection.* The first step of the algorithm is to detect a failure. Because several nodes may detect one failure simultaneously, coordination of concurrent detection is necessary.
2. *Reaching a consistent state.* After a failure was detected, the objects affected by this failure are identified and set to a globally consistent state based on distributed checkpoints.
3. *Computing the new configuration.* Thereafter, the new configuration is computed based on the set of continuously accessable nodes and on the available object data. Moreover, a set of reconfiguration primitives needed for achieving the new configuration is worked out.
4. *Performing the reconfiguration.* The primitives computed in the last step are distributed and executed to set up the new configuration. This configuration may include access limitations in order to ensure consistency.
5. *Resuming application activity.* Finally, the activity of the application is resumed in a way that is consistent with the new state of the objects implied by steps 2 and 4.

This algorithm is performed in each partition. After detecting the recovery of a node or a subnet/physical link, the merging of several partitions is perfomed in the same way, with slight modifications of the first and third step. The steps of the algorithm are described in further detail in section 3.3.

3 Technical Solution

This section describes important details of our technical solution, namely the object placement algorithm, the flexible replication policy, parts of the consistency support, and details of our overall reconfiguration algorithm. The results are also applied to our example application by outlining a concrete dynamic reconfiguration after failure.

3.1 Mapping of the Application Configuration to the Distributed System

The application configuration has to be mapped to the underlying system configuration initially, as well as after a dynamic reconfiguration. This mapping process is guided by logical resource requirements and other placement-related conditions discussed below.

Nodes provide a set of logical resources. Those can be representations of real resources, like printers or mathematical processors, or pure logical resources, introduced for the sake of modelling administrative restrictions. On the other hand, objects need direct access to a set of resources. They can only be located on nodes providing this set. In addition, an object can call for further resources that aren't needed for the basic functionality. For example, an object can claim desired resources for purposes of efficiency. Finally, an object can give a set of direct placement priorities that can be used to state certain administrative recommendations.

For our example, the nodes in figure 1 provide the resources in table 1 and the objects in figure 2 demand or additionally call for these resources as indicated in in table 2; this table also includes direct placement priorities of the objects. Both of the front–end objects for the departments (HD1F, HD2F) use logical resources (D1, D2) to make sure that they are only placed on nodes in their department. The same holds for the front–end object of the personnel department, that use a unique logical resource to make sure that it is placed on node 8.

Node	Provided Resources
1	Unix, math_proc, D1
2	Unix, math_proc, D1
3	Unix, D1
4	VMS, D2
5	VMS, D2
6	Unix, math_proc, D2
7	Unix, math_proc
8	Unix, $P

Table 1: Outline of the system configuration

Object	Required Resources	Desired Resources	Placement Priority
PF	$P, Unix		
PD	Unix		$7 \prec 8$
S	Unix	math_proc	$7 \prec 8$
HD1F	Unix, D1		1
HD2F	VMS, D2		4

Table 2: Outline of the application configuration

In addition, *collocations* between objects can be defined. A collocation defines that its member objects should be placed together on one node. This is useful if a set of objects interacts frequently, because local communication is far more efficient than remote communication. Table 3 shows a collocation that states that the objects PD and S should be placed on one node.

The mapping is realized through a penalty heuristics. Penalties are given for the obstruction of placement priorities, for unfulfilled collocations, and for desired but not granted resources. With one penalty point for an unfulfilled collocation, two points for each violated placement priority and three points for a desired, but not granted resource, the penalty table 4 results for our example. Impossible placements (if mandatory resources cannot be granted) get an infinite penalty value. If a placement priority is defined for an object, nodes not mentioned in the priority list of that object get a priority that is exactly below the last defined one. Associated with the assigmnent of object S to node 3, there is 1 penalty point for the unfulfilled collocation, 3 points for the desired but not granted math_proc and 4 points for the violated priority levels 1 (consisting of node 7) and 2 (node 8). The penalty points for unfulfilled collocations are given to each assignment of an object, as all combinations of specified collocations are seperately computed, if the combination of collocations is possible. For example, if the two collocations {PF, PD} and {PD, HD1F} were defined, each could be fulfilled, but the combined collocation {PF, PD, HD1F} is impossible, as no node provides the resources $P and D1 which are both mandatory for the combination.

Based on the penalty values, the placement of the objects is computed. Each collocation or object is assigned to the node with the lowest penalty value. In the first steps, penalties for placements using collocations are computed, eventually using multiple collocations. After that, a placement without collocations is tested, too. Each of the assignments gets a total penalty value. The assignmnt with the

Collocation	Set of objects
Col1	{PD, S}

Table 3: The defined collocations

Object	Node 1	2	3	4	5	6	7	8
PF	∞	∞	∞	∞	∞	∞	∞	0
PD	5	5	5	∞	∞	5	1	3
S	5	5	8	∞	∞	5	1	6
HD1F	0	2	2	∞	∞	∞	∞	∞
HD2F	∞	∞	∞	0	2	∞	∞	∞
Col2	8	8	11	∞	∞	8	0	7

Table 4: The resulting penalty values

	remaining objects	further collocations	assign- ment	total penalties	
Col1	PF, HD1F, HD2F	%	$Col1 \to 7$ $PF \to 8$ $HD1F \to 1$ $HD2F \to 2$	0 0 0 0	0
%	PD, PF, S, HD1F, HD2F	%	$PD \to 8$ $PF \to 7$ $S \to 7$ $HD1F \to 1$ $HD2F \to 4$	0 1 1 0 0	2

Figure 3: Computed assignments

lowest total penalty value is chosen. The computation of our example (consisting of only two steps here) is outlined in figure 3. The resulting assignment is shown in figure 4.

3.2 Achieving Flexibility of Fault Tolerance

The failures the system can tolerate are determined by the location of the so-called *replication instances* that replicate critical data in the network. This placement should be implied by the kinds of failures a specific distributed system exhibits. Due to the fact that different parts of the application require different degrees of fault tolerance, a complete replication of all critical data exhibits undesirable overhead during normal operation. Considering different degrees of availability for different parts of the application reduces this overhead significantly. Finally, the problem of duplicated objects in disjunctive partitions has to be treated. This is done by restricting the functionality of parts of the application under consideration of administrative requirements.

3.2.1 Placing Replication Instances in the Network

The placement of replication instances (furtheron called RIs) is implied by the definition of the set of failures that has to be tolerated in the worst case (as an example). If a failure happens that isn't covered by the defined set, nevertheless no unrepairable inconsistency is permitted. In such a case the activity of the application is suspended in affected partitions until failed components recover.

Assuming the given system satisfies the fail–stop fault model, any failure can be seen as a partitioning. Consequently, a *specific fault* scenario that has to be tolerated can be defined by a set of potential partitions. Getting back to the distributed system in figure 1, we want the following fault sceanrio to be tolerated in the worst case. Any node, LAN C, and the backbone may fail. If another failure

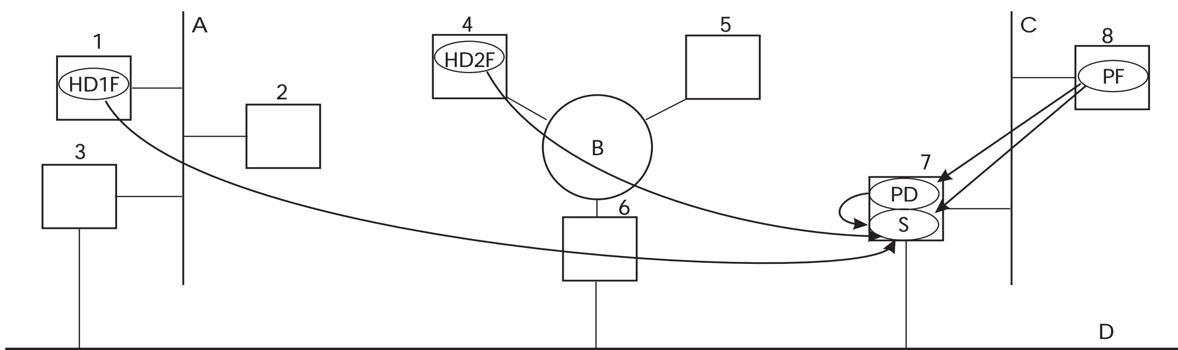

Figure 4: Initial application configuration

Failure of component	Resulting set of partitions
1	{2, 3, 4, 5, 6, 7, 8}
2	{1, 3, 4, 5, 6, 7, 8}
3	{1, 2}, {4, 5, 6, 7, 8}
4	{1, 2, 3, 5, 6, 7, 8}
5	{1, 2, 3, 4, 6, 7, 8}
6	{1, 2, 3, 7, 8}, {4, 5}
7	{8} {1, 2, 3, 4, 5, 6}
8	{1, 2, 3, 4, 5, 6, 7}
C	{8}, {1, 2, 3, 4, 5, 6, 7}
D	{1, 2, 3}, {4, 5, 6}, {7, 8}

Table 5: Using potential partitions to describe a fault scenario

occurs before the handling of a failure is completed, the system is allowed to suspend the activity in some partitions; this case is assumed as being relatively unlikely. The fault scenario implies the set of potential partitions in table 5. Based on this set, RIs are allocated in a way that each potential partition contains at least one RI. As the computation of the minimal set of RIs that fulfills the stated property is NP-hard, we use a heuristics to allocate the RIs. For the given fault scenario, this heuristics leads to the RI locations at nodes 1, 4 and 8. If a failure occurs, the application activity in each resulting partition that contains an RI can continue. After a failure, the same algorithm is executed in each affected partiton, leading to a new set of RIs in each partition. If a resulting partition does't contain an RI, the system activity is blocked in this partition.

The set of potential partitions used to describe a fault scenario may exhibit exponential growth in the number of nodes the network contains. To avoid this undesirable behaviour, all potential partitions that are supersets of other partitions can be excluded from the set of potential partitions. Applying this strategy to the fault scenario we want to tolerate in the example, the set of potential partitions outlined in table 6 results. This leads to the same set of RIs.

{8}
{1, 2}
{4, 5}

Table 6: The reduced set of potential partitions

Appl. functionality on object	Application group
PF	{PF, PD, S}
HD1F	{HD1F, S}
HD2F	{HD2F, S}

Table 7: The notion of application functionalities and groups

A further improvement of this scheme would be possible by introducing a hierarchical subnet structure. In this case, the computations outlined above could first be performed for each subnet separately, before integrating them by treating each subnet as a node during global computation.

The set of potential partitions and the resulting RI locations have to be computed after each failure or recovery of a node or network. The latter is necessary even if all nodes still can communicate, as the failure of a network may reduce the connectivity which could lead to a broader set of potential partitions under the same fault scenario. At the moment, we only provide limited user-level support for specifying the fault scenarios that have to be tolerated; explicit program-level specifications by the system manager are required.

3.2.2 Avoiding complete replication

An object can be allocated in a partition during reconfiguration after failure or recovery if this partition contains an RI that replicates the state of the object. Those replicating RIs have to be chosen under the consideration of specific availability degrees that objects should exhibit.

Our strategy to select the RIs for replicating particular objects is based on the notion of an *application functionality*, i.e. some operation that is directly invoked by an end user and represents a service provided by the distributed application. Each application functionality leads to an application group that is defined as the set of objects that might be invoked by calling an operation on an object via an application functionality, i.e. an application group is the set of objects implementing the application functionality directly or indirectly. Those groups can be inferred from the logical object links specified in the application configuration (section 2.3).

With the application outlined in figure 2 and with application functionalities defined for all front end objects (HD1F, HD2F, PF), the application groups in table 7 result.

Each of the potential partitions contains a number of objects of each application group. Avoiding complete replication now works as follows: Let n be the number of objects in that potential partition that has the maximum number of objects among all potential partitions for a given application group. Then the critical data for the objects of this group are replicated in an RI in each potential partition for which the number of objects of the group in this partition is greater than a predefined percentage of n. Each application functionality is attributed with such a percentage. If no percentage is defined, the default percentage of 100% is used; this percentage leads to replication in just those potential partitions that contain n objects, hence at least in the potential partition from which n is computed. A percentage of 0% leads to complete replication of the objects of the application group, as any potential partition contains at least 0% of n objects. Furtheron we call this algorithm the *threshold concept*. Those computations are performed after each failure or recovery for each application functionality. Consequently, the set of RIs replicating objects changes dynamically with failures or recovery.

If we assume for the sake of our example, that each front–end object carries an application functionality and no threshold percentages are explicitly defined, the default percentage of 100 % is used and the RIs as indicated in table 8 that have to replicate the state of the application groups result. This leads to the locations of replication for the objects, presented in table 9.

With possible extensions towards dynamic code installation (lifting the restrictions of section 1), the

Appl. functionality on object	Replicating RIs
HD1F	1
HD2F	4
PF	8

Table 8: RIs replicating state of the application groups

Object	Replicating RIs
PF	8
PD	8
S	1, 4, 8
HD1F	1
HD2F	4

Table 9: RIs replicating state of the objects

best approach is to replicate code at the same RIs where the associated objects are replicated.

The major advantage of this approach is, that replication is coupled to the notion of application groups, not to isolated objects. Hence the states of all objects that jointly provide an application functionality are available in those RIs that are determined according to the threshold concept. However, the described approach has some limitations: it uses the complete set of potential partitions to compute the desired set of RIs. This set might become rather large if the system grows. But as the stated computations only have to take place after failures or recoveries, this limitation isn't a severe restriction in practice.

3.2.3 Avoiding Inconsistencies among Partitions

The configuration management system may allocate duplicates of objects in different partitions. Unlimited access to duplicates has to be prohibited to enable automatic merging of partitions after failure recovery.

To achieve this, we use the notion of a primary partition for each dedicated set of objects described by a so-called *mark*. Exactly one mark is assigned to each object, the same mark can be assigned to different objects to group them. Unlimited access to an object is only possible in the partition that is the primary partition for the mark of the object. Computing primary partitions is done with an enhanced version of the Voting-Class replication control scheme developed by Tang [TAN90]. The entities that are attributed with weights are the nodes not the RIs. Each node gets a specific weight for each mark. The cited scheme performs dynamic adjustment of the cumulated votes necessary to get a majority, according to the initial vote assignment and the number of votes that participated in the previous majority. Our enhancement provides additional flexibility to manage the dynamic evolution of the system.

To keep the job of an administrator manageable, marks don't have to be specified for each object. Instead marks are assigned explicitly to application functionalities and are propagated along the logical links between objects, as defined in the application configuration. This may lead to the collision of marks on objects, that are used by more than one application functionality. To handle this correctly, priorities between marks can be defined. If no mark is assigned to an application functionality, the mark *sys_default* will be assigned automatically.

Assume the marks P, D1 and D2 are defined for our example, together with the priorities P $\prec$ D1 and P $\prec$ D2. That leads to the priority graph in figure 5. Each user defined mark has a higher priority than

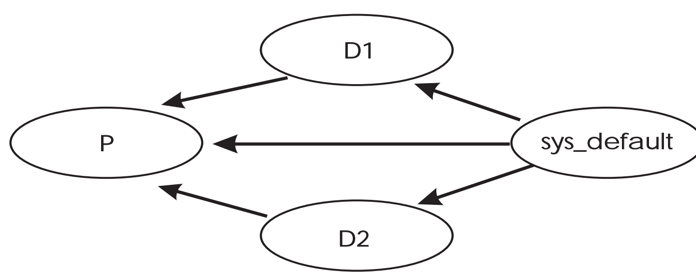

Figure 5: The priorities between marks

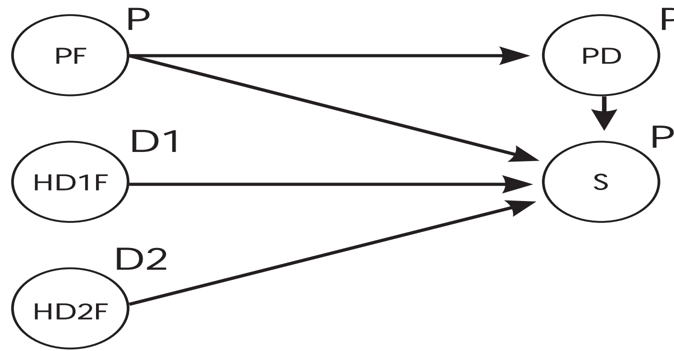

Figure 6: The resulting assignment of marks to objects

sys_default. The graph has to remain acyclic, so that it can be transferred into a linear ordering through topological sorting. One possible result of this sorting is the ordering P $\prec$ D1 $\prec$ D2 $\prec$ sys_default. If D1 is assigned to the application functionality on object HD1F, D2 to the one on HD2F and P to the one on PF, the assignment of marks to objects in figure 6 results.

Marks can be used to handle administrative requirements. If only the nodes of department 1 are weighted with positive votes for mark D1 and analogous weighting is performed for the other marks and departments, the front end objects will exhibit unlimited functionality in their departments if a partitioning occurs. Object S will exhibit the intended unlimited behaviour in the personnel department, whilst duplicates are allocated in the other departments; however, they only support lookup invocations. The latter behaviour is achieved with the specification of the stated priorities between marks. If we access of department 1 to objects used by more than one department is more important than access of the personnel department to those shared objects (priority D1 $\prec$ P), object S would have been marked with D1. It consequently would exhibit unlimited behaviour at the nodes of department 1 in case of a partitioning. If we had modeled the application with multiple storage objects S_i (as mentioned in section 2.2) and specified that the marks of the general departments had higher priority as the one of the personnel department, the following scenario would result: Managers could change the ratings of their employees whilst the personnel department could only perform lookup–calls to (eventually old) copies of the storage objects S_i.

Note that restricting access to lookup–calls in all partitions but the primary one isn't sufficient to guarantee one–copy–serializability of threads. An object O_1 with mark M_1 might be available without limitations in a partition P_1 and available for lookup in a partition P_2. An object O_2 with mark M_2 would have the reverse properties. A thread that updates O_1 after having performed a lookup–call to O_2 might succeed in P_1 and a thread that updates O_2 after a lookup–call to O_1 might succeed in P_2. Those threads aren't serializable. This behaviour was introduced with purpose, but can be enhanced to

to objects that are in their primary partition. Moreover, lookup–access to objects that aren't in their primary partition is only allowed for threads that don't comprise update calls and only touch objects for which the last update took place in the same partition (those pure lookup threads can securely 'run in the past' as in the View–approach [ABT89]).

3.3 Dynamic Reconfiguration revisited

After having introduced the basic foundations, we are now in a place to describe the algorithm for dynamic reconfiguration in more detail. The description focuses on the different steps of the algorithm, outlined in section 2.4. More details of steps one, two and five, especially concerning the modifications to the checkpointing scheme of [LIA90], can be found in [SKK93].

3.3.1 Step one

The first step has to detect failures and perform necessary preparations for the subsequent steps. Failures are detected through the direct, timeout–controlled acknowledgement of messages. This "lazy" approach may lead to the situation that a failure is detected in some of the resulting partitions, whilst it isn't detected in the other partitions; this problem is addressed within the next steps. Recovery of physical components is generally detected through watch–dog instances, probing failed network components or special messages send by recovering nodes.

After the detection of a failure or recovery, the extent of the current partition is analyzed through test messages that are also used for the coordination of concurrent activations. Thereafter, the marks for which the current partition is the primary one are computed. At the end of the step, each node in the new partition is informed about the extent of the partition and about the marks for which the partition is the primary one.

3.3.2 Step two and five

Consistent distributed checkpointing is used to achieve a consistent state before the reconfiguration takes place. Our checkpointing scheme is an enhancement of the *Clouds* approach for global consistent checkpointing [LIA90]. Our scheme can be used without a transaction shell and without access to the stack segments of invocations on objects.

The key problem without a transaction shell was to continue the computations in a consistent way. The checkpoint– and rollback–dependencies, introduced in [LIA90] are extended by work– and call–dependencies that reflect the call history of a thread. Those additional dependencies are installed by a thread–call and released by a thread–return. Consequently, they can be used to repeat thread-calls to objects on failed nodes, and to resume threads that have been partially discarded by a rollback operation. The enhancement guarantees exactly–once semantics for update calls and at–least–once semantics for lookup calls.

Renouncing on the access to the stack segments implies that objects have to be passive at the time they are checkpointed; this is guaranteed by a deadlock–free synchronization scheme between checkpoint operations and threads. The scheme is based on the wound–die deadlock avoidance scheme, first described in [RSL78].

We choose global consistent checkpointing instead of independent checkpointing (e.g. the schemes described in [JUV91]) for one major reason: objects on a failed node may be reallocated on a different node if the object state is available in an RI in the new partition. Therefore, we have to spread all data needed by the checkpointing algorithm for rolling back objects (i.e. installing the correct state) to all RIs that are selected for replication according to the threshold concept. As independent checkpointing

to the RIs atomically. That would result in too much message traffic during normal operation.

3.3.3 Step three

The new configuration in a resulting partitioning is computed in the same way as the initial configuration, i.e. through the usage of the configuration mapping.

Unless the system or application configuration is changed, the penalty values used by the configuration mapping remain the same. Consequently, they needen't be computed again after a failure. Only the last step of the mapping has to be performed for the subset of the system configuration that corresponds to the current partition.

It may happen that objects aren't placeable in a partition. In this case, dummy objects are allocated in the partition that respond to invocations with defined exeption values. Dummy objects are also allocated for objects for which no valid state is replicated in RIs within the new partition.

In the pure checkpointing mode, the caller is informed about the invocation of a dummy object in order to make specific reactions possible. The required exception handling could be performed in an application-specific way. The future integration of distributed transactions would allow for a fully transparent abort and restart. Analogous protocols are used to handle an update call on an object that isn't in its primary partition.

The computed new configuration is finally compared with the old one, available in any RI in the partition. This comparison leads to a set of change primitives that are then applied to the distributed configuration. Objects that are subject to those primitives are rolled back. These operations are necessary to keep the objects consistent with the replicated states of other objects.

3.3.4 Step four

Step four distributes the reconfiguration primitives. This is done with two different protocols. The first one is used for critical global information, like the current location of RIs, the second one for partition dependent information.

The protocol used for critical information is a standard two phase protocol to guarantee consistent atomic change even if a failure occurs. As this protocol guarantees atomicity, it necessarily contains the problem of potential blocking [SKS83].

The protocol used for partition specific information concerning reconfiguration is a nonblocking protocol that guarantees atomicity in the case of uninterrupted action. If a failure during protocol execution occurs, a consistent state is reached atomically in each new partition. We achieved the nonblocking behaviour in renouncing on consistency between partitions. This is unnecessary for partition specific information. The only thing that matters here is the consistency and atomicity within each resulting partition.

The protocol works in three phases:

Phase 1: A coordinator spreads the whole partition specific information to all RIs and additionally to each node all information that the node has to know for reconfiguration. All RIs write the information tentatively to stable storage and each node tentatively performs reconfiguration. They all answer to the coordinator.

Phase 2: After collecting all replies, the coordinator informs the RIs to make the information permanent. After performing this, the RIs send an acknowledge message to the coordinator.

Phase 3: After collecting the acknowledgements, the coordinator sends a message to each node to switch to the new configuration.

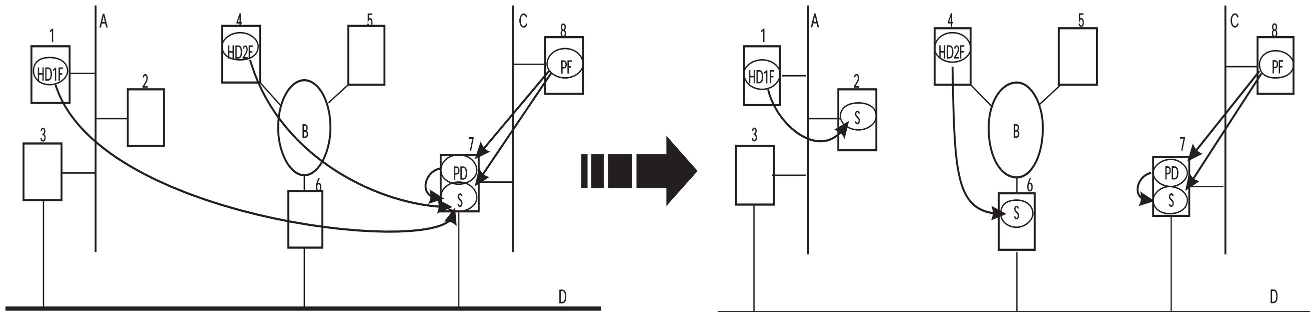

Figure 7: Application structure before and after reconfiguration

In the case of a partitioning, the fate of a resulting partition depends on how far the RIs in it got in the protocol. If one RI in a resulting partition made the new configuration permanent, the new configuration is adopted by all RIs and nodes in the new configuration. If no RI made the new configuration permanent, the tentative information is cleared. Different resulting partitions could come to different decisions, but the decision reached inside each partition is consistent and is the starting point for the reconfiguration that is necessary to adopt to the new partition.

3.4 Example: The Failure of the Backbone Revisited

If the backbone D in figure 1 fails, the system reacts as described below (the initial configuration and the configuration after automatic reconfiguration are shown in figure 7).

We assume that the failure is recognized in each partition (although this is not a necessary precondition). Under the assignment of votes for the department–related marks to the corresponding departments , and the assignment of marks to objects, presented in section 3.2.3, the department nodes are the primary nodes for the corresponding marks. Each resulting partition contains an RI if we use the fault scenario from section 3.2.1. Under the threshold strategy presented in section 3.2.2, the checkpointing information of object S is replicated in the RI of each partition. Data of the front end objects are only replicated in the RIs belonging to the departments, whilst data of object PF is only replicated in the RI of the personnel department.

This leads to the situation that the subconfiguration consisting of the objects PD, PF and S is newly installed in the personnel department, allowing unlimited access to all operations. In the other departments, a configuration is reached that consists of the front end objects of the departments with unlimited access, and of a duplicate of S with restricted access.

Finally, according to the fault–scenario, new sets of potential partitions are computed, new RIs are allocated in the partitions, and the replication strategy of the objects in each partition is adjusted to the new configuration. Finally watch–dog instances are set up that probe the backbone.

4 Initial Experience

The implementation of the overall approach consists of three major components, namely the distributed C++ extension, the general configuration management support, and the different mechanisms for fault tolerance.

Distributed C++ extension: This component (described in detail in [SCH92]) has been implemented as a C++ class library under Unix (Ultrix) on DECStations 5000 and 3100. For low–level network

process are implemented by a modified version of the AT&T C++ threads package.

All distribution–related object management operations are defined within a specific class *DObject* and are inherited by application–specific classes. In addition to remote invocation, basic operations for migration of non–activated objects between nodes and for related locating, fixing and unfixing purposes are supported:

```
class DObject {  // ... instance variables
 public:  DObject (LogicalNode*);       // constructor to create at node
          ~DObject ();                  // destructor
          LogicalNode *locate ();       // to locate the object
          boolean move (LogicalNode*);  // to move the object
          void fix ();                  // to fix the object (avoid move)
          void unfix (); };             // to release the object
```

All objects are identified via globally unique numbers, consisting of their birthnode address and a locally unique identifier. For an object invocation, the global identifier is either mapped to a local address (for local objects), or is mapped to a forwarding address for remote objects; the object can then be located at the indicated node or at a subsequent node.

The current performance is in the order of 10ms for a Null–RPC between two DECStations 5000 via Ethernet; most of the time caused by local protocol handling and object management tasks.

The remote invocation facilities of the C++ extension are used as a base for the distributed configuration management mechanisms.

Distributed configuration management: The configuration language is transformed into the internal representation by a compiler implemented with LEX and YACC. The tool consists of the regular compiler components. The installation of the initial configuration is performed by the initial configuration manager written in C++. The most important task of it is computation of object placement. Our implementation does not use exactly the same scheme as described in section 3.1 but still an earlier, somewhat more complicated version based on a larger set of placement conditions. The new scheme has been developed based on experiences with the former heuristics. All objects declared in the configuration language are mapped to C++ objects of subclasses of *DObject*. They are fixed and therefore are not subject of dynamic migration.

Dynamic configuration change requests are issued by the fault tolerance components via a small procedural interface. They are executed in a decentralized way on different nodes by change managers; the internal management protocol is also based on remote C++ communication.

Fault tolerance support: Of the described fault tolerance components to perform the automatic reconfiguration algorithm, the new approach to map a configuration, the distributed checkpointing scheme and the extended voting class algorithm have been implemented within a specific test environment. This environment allows interception and interactive delay of all remote messages to objects. This way, inherent race conditions and time-dependent problems could be tested in a deterministic way. The relevant protocols for both parts are based on C++ object interactions between objects of several internal auxiliary classes.

The mechanisms for failure detection and the RI related protocols, however, are only emulated in a preliminary way within the test environment.

The configuration management support makes structural management of applications simpler. Most important, the declarative language is quite easy to handle. Additional features like module hierarchies or multilanguage support as found in *Conic* [KMS89] or *REX* [MKS90], respectively, would be desirable but were outside the scope of our work.

Most important, the fault tolerance concept proves that automatic structural reconfiguration of distributed applications is possible, in general, and that a high level of transparency (automatic reaction

5 Related Work

Communication base and object management: Our system model and management approach relies on the RPC-/object interaction paradigm as the pure base of communication. There have been several developments in the field of object–oriented distributed systems (e.g. *Emerald* [JLH88] or *Amadeus* [HOC91]); Our C++-based approach inherits major concepts from these systems and mainly adds configuration management support (see below), as well as extended thread management. As standardized implementations of the RPC paradigm are becoming more popular (especially the OSF DCE [OSF91]), we intend to use their broader functionality instead of using TCP/IP access directly and expect to experience a gain in performance, with using the DCE thread–package. Moreover, fault tolerant communication mechanisms as found in the *ISIS* system [BIJ87] could be used to facilitate a simpler implementation of some of our distributed protocols (e.g. distribution of configuration data).

Distributed configuration management: The best known system providing our notion of configuration management in a very sophisticated way is *Conic* [KMS89]. Compared to Conic, our approach emphasizes automatic handling of failures but does not address advanced structuring concepts like Conic. Achieving fault tolerance with Conic was also realized in a different way [KMY90], [LOK86]; the main difference to our system is that we do not rely on special purpose code tailured to the specific application, but use generic techniques that can be customized to the requirements of a specific environment and application. System support for dynamic reconfiguration, comparable to Conic, is also exhibited by other configuration management systems. All those systems have been designed under different goals (e.g. the integration of modules written in multiple programming languages (*REX* [MKS90], the successor of Conic), or support for system level programming in *Lady* [WYB90]). Extended system support for Fault–Tolerance within a configuration management system is also found in *Durra* [BDW90]. Durra provides an interface for automatic replication of data; however, consistency issues have to be handled by the application code.

Replication control and consistency: The scheme used for replication control is a direct extension of the Voting Class algorithm, described in [TAN90]. We chose this algorithm among all possible voting schemes (e.g. dynamic voting) because of its flexibility. Our extension allows for dynamic evolution of the number of controlled replicas.The functionality our approach currently exhibits can best be compared to the concept of Views [ASC85]. Views can briefly be described as an approximation of the set of available objects, where object replication normally (but not necessarily) follows a read–one, write–all approach. If writing fails or a recovering node is detected, a new view with a new set of accesible objects could be installed (depending on the tracking strategy) with a view–change protocol. This is similar to our approach of a primary paritition for marks, within which all RIs have to be reached to write a checkpoint. Also the idea of using the properties of advanced voting protocols with views is not new. A combination based on an other dynamic voting protocol could be found in [ABD91]. Our novel feature is that we decouple votes from replicas and couple them to nodes. This gives us the necessary flexibilty to allocate or deaalocate RIs even in small non–primary partitions as no votes have to be included or excluded for the price that it may happen in rare cases, that we have a primary partition for some marks, but no current state for the objects attributed with the marks, as not all nodes that carry votes also carriy an RI. The latter case is detected by partition–IDs that are rather compareable to the view–IDs necessary in the View approach.

The platform of our consistency mechanisms is an enhancement of the scheme used in the Clouds system [LIA90]. The extension allows the use of checkpointing to achieve exactly–once semantics for update–calls and at–least–once semantics for lookup–calls. Furthermore this extension allows the use of standard thread–packages as it doesn't rely on the ability to checkpoint the stack–segments of threads. As an additional feature, the scheme can be used as a base to implement a transaction facility, comparable to *TABS* [SBD85] and *Argus* [LIS83]; however, this has not been done by us yet.

An alternative approach for achieving fault tolerance is based on forward recovery using replicated

during normal operation. Fault tolerance support based on dedicated distributed operating system and communication platforms has been implemented in the *Delta-4* [POW91] and *MARS* [KDK89] projects. Such a system-level approach is not within the scope of our work but is an important alternative if special hardware and operating system support is available, or tolerating failures beyond the fail–stop case is of importance. Moreover, a good survey of architectural considerations and mechanisms in the distributed fault tolerance area is found in [JAL94]

6 Conclusion

The report has described a new approach supporting transparent reconfiguration of distributed applications after failures. It is centered around the capabilities of configuration managers in distributed systems. We were only interested in support for replication and checkponting as far, as was necessary for turning the configuration manager in an utility that provides fault–tolerance to an application in a transparent manner. This transparency doesn't comprise the system administrator, who has to specify several configuration details.

The major benefits of the approach are maintenance of application functionality in the case of system failures, provision of consistency of ongoing computations, and flexibility of the level of desired fault tolerance. The latter comprises compliance to the failure characteristics that should be tolerated by the specification of fault scenarios, the degrees of availability for different parts of the application by the properties of our threshold concept and finally the flexible placement of replication instances based on the stated specifications. In addition administrative requirements could be specified, comprising the locations of high availability by the assignment of mark specific votes to selected nodes and availability trade–offs between applications assigning priorities between marks. The technical concept has the major advantage of exploiting the capabilities of a configuration manager to infer the set of objects that have to work together to provide a specific application functionality. Hence the concepts for fault–tolerance aren't tied to isolated objects, but moreover to object groups, providing advanced application-level functionality.

Future work can be divided into completing the implementation on one hand, and into conceptual enhancements on the other. Our implementation work will focus on support for distributed transactions in addition to checkpoints, on extended support for configuration management (e.g. regarding hierarchies of application objects), and into exploiting how far the configuration capabilities of traders [ISO92] could be exploited to be a further base beyond the configuration manager.

References

[ABD91] El Abbadi A., Dani S.N.: A dynamic accessibility protocol for replicated databases; *Data & Knowledge Engineering, 1991, pp. 319-332*

[ABT89] El Abbadi A., Toueg S.: Maintaining Availability in Partitioned Replicated Databases; *ACM Transactions on Database Systems, Vol. 14, No.2, June 1989, pp.264-290*

[ADL90] Ahamad, M., Dasgupta, P., LeBlanc, R.J., Wilkes, C.T.: Fault Tolerant Atomic Computing in an Object-Based Distributed System; *Distributed Computing, Vol. 4, 1990, pp. 69-80*

[ASC85] El Abbadi A., Skeen D., Cristian F.: An efficient fault–tolerant protocol for replicated data management; *Proc. of the 4th International ACM Symposium on Principles of Database Systems, 1985, pp. 215-229*

[BDW90] Barbacci, M.R., Doubleday, D.L., Weinstock, C.B.: Application-Level Programming; *Int. Conf. on Distributed Computing Systems, Paris, 1990, pp. 458-465*

[BIJ87] Birman, K.P., Joseph, T.A.: Reliable Communication in the Presence of Failures; *ACM Trans. on Computer Systems, Vol. 5, No. 1, Feb. 1987, pp. 47-76*

veys, Vol. 23, No. 1, March 1991, pp. 91-124

[HOC91] Horn, C., Cahill, V.: Supporting Distributed Applications in the Amadeus Environment; *Computer Communications, Vol. 14, No. 6, July/Aug. 1991, pp. 358-365*

[IEE92] *Proc. of the Int. Workshop on Configurable Distributed Systems, London, March 1992, Institute of Electrical Engineers (IEE), London*

[ISO92] ISO/IEC JTC1/SC21/WG7: A structural specification of the ODP trader with federating included, 1992

[JAL94] Jalote P. Fault Tplerance in Distributed Systems *Prentice Hall, Englewwod Cliffs, New Jersey, 1994*

[JLH88] Jul, E., Levy, H., Hutchinson, N., Black, A.: Fine-Grained Mobility in the Emerald System; *ACM Transactions on Computer Systems, Vol. 6, No. 1, Feb. 1988, pp. 109-133*

[JUV91] Juang T. T-Y., Venkateson S.: Crash Recovery with little Overhead; *Proc. 11th Int. Conference on Distributed Computer Systems, Arlington, Texas, May 1991, pp. 454-461*

[KDK89] Kopetz, H., Damm, A., Koza, C., Mulazzani, M., Schwabl, W., Senft, C., Zainlinger, R.: Distributed Fault-Tolerant Realtime Systems: The MARS Approach; *IEEE Micro, Feb. 1989, pp. 25-40*

[KMS89] Kramer, J., Magee, J., Sloman, M.: Constructing Distributed Systems in CONIC; *IEEE Trans. on Software Engineering, Vol. 15, No. 6, Juni 1989, pp. 663-675*

[KMY90] Kramer, J., Magee, J., Young, A.: Towards Unifying Fault and Change Management; *2nd IEEE Workshop on Future Trends of Distributed Computing Systems; Cairo, Sept. 1990, pp. 57-63*

[LIA90] Lin L., Ahamad, M.: Checkpointing and Rollback–Recovery in Distributed Object-Based Systems; *20th Symposium on Fault Tolerant Computing, University of North Carolina, June 1990, pp. 97-104*

[LIS83] Liskov, B., Scheifler, R.: Guardians and Actions: Linguistic Support for Robust, Distributed Programs; *ACM Trans. on Programming Languages and Systems, Vol. 5, 1983, pp. 381-404*

[LOK86] Loques, O.G., Kramer, J.: Flexible Fault Tolerance for Distributed Computer Systems; *IEE Proc. Part E: Computers and Digital Techniques, Vol. 133, No. 6, Nov. 1986, pp. 319-332*

[MKS90] Magee, J., Kramer, J., Sloman, M., Dulay, N.: An Overview of the REX Software Architecture; *2nd IEEE Workshop on Future Trends of Distributed Computing Systems, Cairo, Sept. 1990, pp. 396-402*

[OSF91] Open Software Foundation: OSF Distributed Computing Environment - An Overview; *OSF, Cambridge, MA, 1991*

[POW91] Powell, D. (Ed.): Delta-4: A Generic Architecture for Dependable Distributed Computing *Springer–Verlag New York Berlin Heidelberg, 1991*

[RSL78] Rosenkrantz, D.J., Stearns, R.E., Lewis II, P.M.: System Level Concurrency Control for Distributed Databases; *ACM Trans. on Database Systems, Vol. 3, No. 2, June 1978, pp. 178-198*

[SBD85] Spector, A.Z., Butcher, J., Daniels, D.S., Duchamp, D.J., Eppinger, J.L., Fineman, C.E., Heddaya, A., Schwartz, P.M.: Support for Distributed Transactions in the TABS Prototype; *IEEE Trans. on Software Engineering, Vol. 11, 1986, pp. 520-530*

[SCH92] Schill, A.: Distributed Object Management within a Loosely-Coupled Repository Environment; *OpenForum Technical Conf., Utrecht, Nov. 1992*

[SCS83] Schlichting, R.D., Schneider, F.: Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems; *ACM Trans. on Computer Systems, Vol. 3, No. 1, Feb. 1983, pp. 15-30*

[SCZ93] Schill, A., Zitterbart, M.: A System Framework for Open Distributed Processing; *Journal of Network and Systems Management, Vol. 1, No. 1, 1993*

[SKK93] Schill A., Kottmann D., Keller L.: Fehlertoleranz durch dynamische Rekonfiguration verteilter Anwendungen (Fault Tolerance by Dynamic Reconfiguration of Distributed Applications); *Proc. ITG/GI-Fachtagung Kommunikation in verteilten Systemen (ITG/GI-Conference Communication in Distributed Systems), March 1993, Munich, Informatik-Fachberichte, Springer, pp. 340-354*

on Software Engineering, Vol.SE-9, No.3, May 1983, pp. 219-228

[TAN90] Tang, J.: Voting Class – an Approach to Achieving High Availability for Replicated Data; *2nd International Symposium on Databases in Parallel and Distributed Systems, Dublin 1990, pp. 146-156*

[WYB90] Wybranietz, D., Buhler, P.: The LADY Programming Environment for Distributed Operating Systems; *Future Generation Computer Systems, Vol. 6, No. 3, Dez. 1990, pp. 209-223*