Bridging the Gap between Hardware and Software Fault Tolerance

M. Patiño-Martínez R. Jiménez-Peris
Technical University of Madrid (UPM)
Facultad de Informática
E-28660 Boadilla del Monte, Madrid, Spain
{mpatino, rjimenez}@fi.upm.es

A. Romanovsky
University of Newcastle upon Tyne
Department of Computing Science
NE1 7RU Newcastle, England
Alexander.Romanovsky@newcastle.ac.uk

Abstract

During the last decades several mechanisms for tolerating errors caused by software (design) faults have been put forward. Unfortunately only few experimental programming languages have incorporated them, so these schemes are not available in programming languages and systems that are used in developing modern applications. This is why programmers must either implement these mechanisms themselves or follow very complicated guidelines. It is not the case for software mechanisms developed for tolerating hardware faults (site crashes). Many programming languages and development systems provide mechanisms to cope with site failures. For instance, transactions are defined as one of the basic services in CORBA, Enterprise JavaBeans and Jini, the most popular middleware platforms used for developing complex distributed applications. In this paper we demonstrate how to implement recovery blocks and N-version programming, the most popular mechanisms developed for tolerating software errors, on the top of the mechanisms proposed for tolerating hardware errors.

Keywords: recovery blocks, n-version programming, exception handling, transactions, software and hardware fault tolerance.

Wordcount: ≈ 6100

Submission category: Regular paper

Bridging the Gap between Hardware and Software Fault Tolerance *

M. Patiño-Martínez† R. Jiménez-Peris† A. Romanovsky‡

† Technical University of Madrid (UPM), Facultad de Informática,
E-28660 Boadilla del Monte, Madrid, Spain, {rjimenez, mpatino}@fi.upm.es
‡ University of Newcastle upon Tyne, Department of Computing Science, NE1 7RU
Newcastle, England, Alexander.Romanovsky@newcastle.ac.uk

Abstract

During the last decades several mechanisms for tolerating errors caused by software (design) faults have been put forward. Unfortunately only few experimental programming languages have incorporated them, so these schemes are not available in programming languages and systems that are used in developing modern applications. This is why programmers must either implement these mechanisms themselves or follow very complicated guidelines. It is not the case for software mechanisms developed for tolerating hardware faults (site crashes). Many programming languages and development systems provide mechanisms to cope with site failures. For instance, transactions are defined as one of the basic services in CORBA, Enterprise JavaBeans and Jini, the most popular middleware platforms used for developing complex distributed applications. In this paper we demonstrate how to implement recovery blocks and N-version programming, the most popular mechanisms developed for tolerating software errors, on the top of the mechanisms proposed for tolerating hardware errors.

Keywords: recovery blocks, n-version programming, exception handling, transactions, software and hardware fault tolerance.

1 Introduction

Considerable research has been carried out in the field of fault tolerance. As a result, many schemes and techniques have been developed for dealing with both hardware and software faults. With the increasing use of distributed systems, many of the software techniques for tolerating (site) hardware faults have been incorporated into a number of software development kits. In contrast, the use of software fault tolerance is not particularly widespread. Paper [LABK90] advocates the integrated use

*This research has been partially funded by the Spanish National Research Council, CICYT, under grant TIC2001-1586-C03-02, and by European IST DSoS project (IST-1999-11585).

of hardware and software fault-tolerance mechanisms to effectively tolerate faults of any nature. Later [Jal94] suggested that the basic mechanisms for hardware fault tolerance could be used to implement software fault tolerance mechanisms.

Software fault tolerance mechanisms employ two main approaches to detecting errors caused by design faults [LABK90]: (1) acceptance tests and (2) diversified design. Acceptance tests are executable post-conditions that check the validity of the results of a service. Diversified design produces several implementations, or *variants*, meeting the same service specification. Based on these basic elements, several techniques have been proposed in literature, the most popular of which are recovery blocks [Ran75] and N-version programming [Avi85]. Recovery blocks are a backward error recovery scheme relying on state restoration features, whereas N-version programming makes use of parallel execution of versions.

Exception handling is another fault tolerance technique [Cri89] that allows programmers to tolerate faults of several categories such as environmental faults, operators' mistakes, faults of the underlying hardware and software, etc., at the application level. Exception handling is widely used as most modern programming languages incorporate it and have features for declaring exceptions in different scopes and associating handlers with them. In general, exceptions of each scope are divided into *external* and *internal* [XRR98, Rom00a], depending on whether they are propagated out of the scope where they are signalled or not. Exceptions handled in the scope where they are *raised* are classified as internal exceptions. Exceptions that are propagated outside the scope in which they are *signalled* are called external exceptions.

Software and hardware fault tolerance techniques employ a number of common or similar features. There is a number of hardware fault tolerance schemes relying either on returning the system to a previous state or on parallel/distributed execution of redundant software. Transactions belonging to backward error recovery preserve data consistency in the presence of site failures by returning the system into a previous state: if a site crashes, the transaction aborts and its effect is undone, data are taken back to the state they were in before the transaction started. If an alternate fails to ensure the acceptance test, the state restoration features employed in recovery blocks return the system into a previous state before trying the next alternate.

Group communication (multicast and membership service) is widely used in developing distributed applications. One of the main areas in which it is employed is active replication. A process (or object) is replicated and executed on different machines to cope with site failures. The multicast hides from the client the fact that there are several replicated processes running. But this is not the only application of group communication. Processes can have a different code but share a common interface and several services can be run in parallel. We can compare this use of group communication with N-version programming, which consists in running in parallel several programs with a common specification. The results are obtained by voting. The fact that a program component is diversely designed and N versions are run in parallel is hidden.

In this paper we explore similarities in some of the software and hardware fault tolerance techniques. The contribution of the paper is that it proposes ways of implementing software fault tolerance

on top of hardware fault tolerance techniques. The applicability of these ideas is demonstrated with a distributed programming language, *Transactional Drago*, supporting transactions and group communication. Although we use a particular programming language, *Transactional Drago*, our analysis shows that minimal modifications will be required to allow the same approaches to be used in building software fault tolerance on top of current middleware systems. For example, support for transactions is available in CORBA [OMG], Jini [AOS+99], and Enterprise Javabeans [Sun]. Moreover, a group service has been recently included in the Fault Tolerant CORBA specification [OMG00]. Most of the behaviour we assume *Transactional Drago* provides for groups is part of this specification or will soon be included. We assume that other middleware technology will follow a similar approach to that of the Fault Tolerant CORBA.

The paper starts with describing the system model and continues with a brief introduction to *Transactional Drago* - an Ada extension for developing distributed transactional applications. In the following parts we discuss how recovery blocks and N-version programming can be applied in distributed client-server settings to tolerate hardware and software design faults, and demonstrate these ideas using *Transactional Drago*. Apart from this, sections 4 and 5 show how these two software fault tolerance mechanisms can be extended to accommodate systematic exception handling and discusses possible approaches to implementing these proposals in *Transactional Drago*. The paper finishes with the discussion and conclusions sections.

2 Model and Definitions

2.1 System Model

The system consists of a set of sites (nodes) $S = \{S_1, S_2, ..., S_N\}$ provided with stable memory that communicate by exchanging messages through reliable channels. We assume an asynchronous system where nodes fail by crashing (no Byzantine failures). Failed sites may recover with its stable memory intact.

2.2 Communication Model

Sites are provided with a group communication system supporting strong virtual synchrony [FvR95]. Group communication systems provide reliable multicast and group membership services. Group membership services provide the notion of view (current connected and active processes). Changes in the composition of a view (addition or deletion) are delivered to the application. We assume a primary component membership [CKV01]. In a primary component membership, views installed by all processes of a group are totally ordered (there are no concurrent views), and for every pair of consecutive views there is at least one process that survives from the one view to the next one.

Strong virtual synchrony ensures that messages are delivered in the same view they were sent and that two processes transiting to a new view have delivered the same set of messages in the previous view.

Regarding ordering guarantees of multicast messages, we will use total order multicast, which ensures that messages are delivered in the same order at all group members.

We will assume a client/server model of interaction. A server is a group of processes that provides a set of remotely callable services (methods), which must be implemented inside each group process. Clients communicate with servers using group communication primitives. That is, service requests are multicast to all the processes of a group in total order.

2.3 Transactions

A transaction is a sequence of operations that are executed atomically, that is, they are all executed (the transaction commits) or the result is as if none of them had been executed (it aborts). Two operations on the same data item conflict if they belong to different transactions and at least one of them modifies the data item. Transactions with conflicting operations must be isolated from each other to guarantee serializable executions [BHG87].

Transactions can be nested [Mos85]. Nested transactions or *subtransactions* can be executed concurrently, but isolated from each other. Transactions that are not nested inside another transaction are called *top-level transactions*. If a top level transaction aborts, all its subtransactions and their descendants will also abort, no matter whether they have committed or aborted. However, a subtransaction abortion does not compromise the result of its parent transaction (the enclosing one). Hence, subtransactions allow failure confinement.

3 *Transactional Drago*

Transactional Drago [PJA98] is an extension to Ada [Ada95] for programming distributed transactional applications. *Transactional Drago* implements *group transactions* [PJA02], a new transaction model that supports multithreaded transactions. As a result it is possible to take advantage of the multiprocessor and multiprogramming capabilities.

Programmers can start transactions using the begin-end transaction statement or *transactional block*. Transactional blocks are similar to block statements in Ada. They have a declarative section, a body and can have exception handlers. The only difference with block statements is that the statements inside a transactional block are executed within a transaction. A transactional block can be enclosed within another transactional block leading to a nested transaction structure. Any unhandled exception in a transaction aborts the transaction. The exception is signalled in the transaction enclosing scope.

All data used in transactional blocks are subject to concurrency control (in particular, locks) and are recoverable. That is, if the corresponding transaction aborts, data will be restored to the value they had before executing that transaction. Data items can be volatile or non-volatile (persistent). Concurrency control mechanisms are implicitly handled by the run-time system [PJKA02], hiding all the complexity from the application programmer. Programs access transactional data (atomic data) just as regular non-transactional data. Programmers do not set and free locks. The underlying system

is in charge of ensuring the isolation and atomicity properties of the data.

Processes are the unit of distribution in *Transactional Drago*. Processes belong to groups. A group is seen as an individual logical entity, which does not allow its clients either to view its internal state, nor the interactions among its members. Processes belonging to the same group share a common interface and application semantics. A group interface is a description of remotely callable services, which must be implemented inside each group member, and other information available to clients of the group like exceptions the group can signal.

Group communication primitives are used to communicate with groups. A request to a group is multicast in total order to all the processes of a group. We distinguish two kinds of groups, *replicated* and *cooperative* groups, according to the state and behavior of its members.

Replicated groups implement the active replication model, that is, they behave as state machines [Sch90]. According to this model all the group members are identical replicas, that is, they have the same state and code, and should run on failure-independent sites. Group members must behave deterministically. Therefore, since all group members receive the same requests in the same order (Fig. 1.b), they will produce the same answers. The results of the replicas are filtered, so that the results of a single replica are returned to the client (transparent replication).

A replicated group can act as a client of another group. Replication transparency is provided by the underlying communication system, *Group_IO* [GMAA97], that filters the replicated requests so that a single message is issued (Fig. 1.c). This type of communication allows building programs with active replication and minimal additional effort from the programmer. That is, the programmer programs the group as if the group were made out of a single process.

On the other hand, members of a *cooperative group* do not need to have either the same state or the same code. They are intended to divide data among its members and/or to express parallelism taking advantage of multiprocessing or distribution capabilities in order to increase the throughput. Members of a cooperative group are aware of each other and they can communicate by multicasting messages to the group. This kind of communication is called *intragroup* communication. Invocations to or from a cooperative group are independent so they are not filtered by the communication system (Fig. 1.a). The client (server) receives a reply (request) per group member.

Transactional Drago provides support for exception handling in both kind of groups [PJA01]. The interface of a group can include the exceptions the group can signal. In replicated groups, all members are supposed to signal the same exception, if it is not the case, members that disagree are considered faulty. Members of a cooperative group can finish a service signalling different exceptions (they have different code). In such a situation, by default, the predefined exception `severalExceptions` is signalled to the client. The group programmer can overwrite this behaviour defining an exception resolution function for each group service.

If a group is called and some members are available but, there is no primary component, the predefined exception `group_error` is signalled to the client.

In order to provide an adequate level of availability failed group members will join the group they belong when they are available. A recovered member will need to bring its state up to date before start-

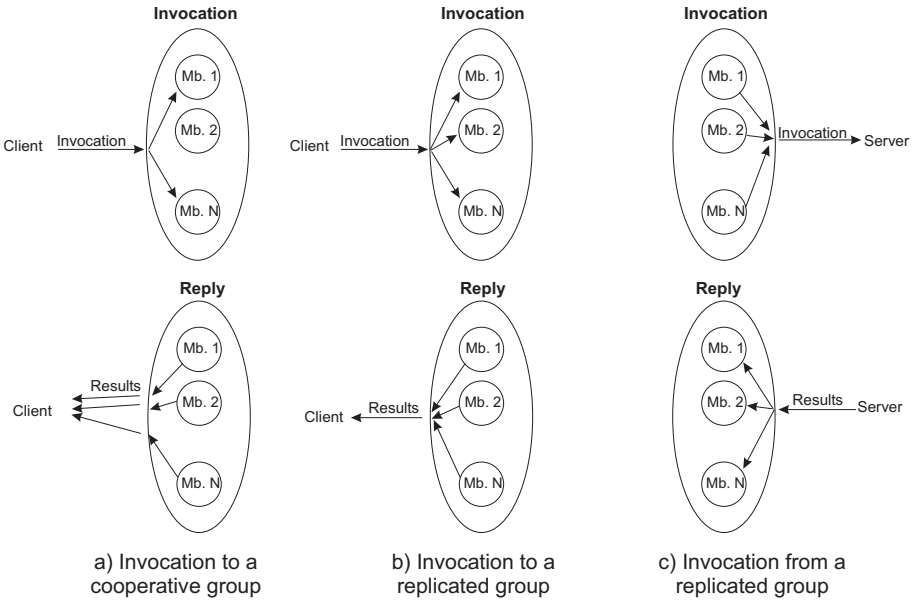

Figure 1: Group invocations and invocations from groups

ing processing group request. This recovery process is automatically performed in replicated groups, since all group members have the same state and code. However, this is not the case of cooperative groups. The group programmer must provide a recovery function for members of cooperative groups.

4 Recovery Blocks

4.1 The Basic Mechanism

The recovery block [Ran75] scheme is based on software redundancy and rollback to recovery points. A number of variants (*alternates*) are independently produced from the same specification by different designers (maybe by using different algorithms). Besides, an acceptance test is to be implemented to check the correctness of the alternate results. A recovery point is established when the program enters the recovery block. The primary alternate is executed and the acceptance test checks its correctness. If it fails, the program is restored to the recovery point, and the next alternate is tried. This sequence continues until either the acceptance test is ensured or all alternates have been tried and failed. In the first case, the recovery point is discarded, and the block is exited. In the second one, the recovery point is restored (to keep the program in a predefined known state), and a failure exception is signalled. Recovery blocks can be nested, in which case the results of the nested recovery block must be flushed away if the alternate containing it has not passed its acceptance test. Note that recovery blocks and transactions belong to backward error recovery schemes, and that there are certain similarities in their behaviour when the acceptance test fails and a transaction aborts because the system state is rolled back to a previous state.

4.2 Extensions of the Mechanism for Exception Handling

The original proposal of recovery blocks does not introduce a full-fledged exception handling mechanism: the use of exception handling here is restricted to only signalling the failure exception when a recovery block fails to produce the results, and to handling any exception by interrupting the current alternate, restoring the previous state and trying another alternate. We believe that recovery block scheme can benefit from introducing a general exception handling mechanism. First of all, this will allow any alternate to have a number of internal exceptions declared by the alternate developer (with the corresponding handlers), so that when such an exception is signalled the corresponding handler inside the alternate will try to handle it locally. If the exception has been handled successfully the acceptance test is checked and the execution of the recovery block proceeds. If an external exception is signalled outside the alternate (this can be done, for example, by a handler of an internal exception), the system state is restored by the recovery block support and another alternate is tried. This approach seems to be the most practical one, although it may raise some doubts because we abort the system state irrespective of the external exception signalled. One of the advanced solutions here could be to allow recovery blocks to have several outcomes: one normal, several exceptional and a failure. Exceptional outcomes will be associated with external exceptions and each of those outcomes will have a special acceptance test associated with it. If the acceptance test is passed, the alternate succeeds and the exception is signalled. If it fails, the next alternate is executed. For example, we can extend the recovery block to allow for two external exceptions: `buffer_empty` and `buffer_full`, in which case if an alternate signals one of these exceptions the corresponding acceptance test is checked: if it is satisfied we signal the exception outside, otherwise the state is restored to the previous one and the next alternate is tried.

Paper [MSR77] puts forward the idea of combined use of recovery blocks and exceptions, so that the first scheme would allow tolerating design faults whereas the second will be used for dealing with anticipated faults inside alternates.

4.3 Implementations of Recovery Blocks with Transactions

We will first show how to implement a recovery block in which external exceptions cause state restoration. Then, we will extend that implementation to consider external exceptions as a valid result of an alternate.

To simplify the code each alternate is encapsulated in a procedure, `alternateX`, which includes the handlers for its internal exceptions. The recovery block tries sequentially its constituent alternates. This behaviour can be modelled by using a loop that tries one alternate at a time until either an alternate succeeds or all the alternates have been executed. The execution of an alternate must be such that if the acceptance test fails, the state is restored. This behaviour is automatically provided by enclosing both the alternate and the acceptance test in a transaction. The loop contains the transaction that encapsulates the execution of the current alternate.

After the alternate execution, the acceptance test will be executed to check the correctness of the

variant. If the acceptance test fails, an exception is raised (signalled using Ada terminology) to cause the transaction abort and therefore, rollback of the state.

The transaction is enclosed within a block statement which provides an exceptional context. If an exception is propagated out of an alternate or is signalled in the acceptance test, the transaction aborts and control is transferred to the exception handlers of the block statement. The exception handler (associated to the block statement) captures every exception (`when others`) and forces the execution of the next alternate, if any. If there are no more alternates the recovery block ends by raising a `failure` exception. The recovery block can be written in *Transactional Drago* as follows:

```
ok:= false;
alternate:= 1;
while alternate <= N and not ok loop
  begin -- block that captures unhandled exceptions in the transaction
    begin transaction
      case alternate of
        1: alternate1(params);
        2: alternate2(params);
        ...
      end case;
      ok:= acceptance_test(params);
      if not ok then
        -- the acceptance test has failed, abort the transaction
        raise failure;
      end if;
    end transaction;
  exception -- current alternate has failed, try the next one
    when others =>
      -- the alternate has failed, try the next one if any
      if alternate < N then
        alternate:= alternate +1;
      else
        raise failure;
      end if;
  end;
end loop;
```

Any unhandled exception in an alternate will abort the transaction and therefore, the state will be restored to the previous one. The exception will be handled in the `others` handler and the next alternate will be executed.

If an alternate contains a recovery block, the alternate must contain the previous code, yielding to a nested recovery block (transaction) structure.

The previous code must be modified to associate different acceptance tests with external exceptions. If an alternate signals some external exception, and the associated acceptance test is passed, the alternate succeeds and that exception is propagated out of the recovery block. To achieve this behaviour, those exceptions must be handled inside the transaction, otherwise the transaction will abort and the state rolled back. Therefore, we need an additional exception context (block statement) within the transaction to capture those exceptions. Otherwise, an (unhandled) external exception will abort the transaction and therefore, the alternate, and the state will be rolled back. The block has a handler

for each external exception, which contains the associated acceptance test. If the acceptance test succeeds, the exception is saved to be signalled again after the transaction has committed (the alternate has succeeded). If the acceptance test fails, the `failure` exception is raised, the transaction aborts and the next alternate executed.

```
raise_exception:= false;
ok:= false;
alternate:= 1;
while alternate <= N and not ok loop
  begin -- block that captures unhandled exceptions in the transaction
    begin transaction
      begin -- block to capture external exceptions
        case alternate of
          1: alternate1(params);
          2: alternate2(params);
          ...
        end case;
        ok:= acceptance_test(params);
        if not ok then
          raise failure;
        end if;
      exception
        -- an alternate or the acceptance test have raised
        -- an exception, execute the associated acceptance
        -- test if it is an external exception
        when excep: external1 =>
          ok:= acceptance_test_external1(params);
          -- if the acceptance test has failed, abort
          -- the transaction. Execute the next alternate
          if not ok then
            raise failure;
          else
            raise_exception:= true;
            Save_Ocurrence(external_exception, excep);
          end if;
        when excep: external2 =>
          ok:= acceptance_test_external2(params);
          if not ok then
            raise failure;
          else
            raise_exception:= true;
            Save_Ocurrence(external_exception, excep);
          end if;
      end; -- handler for external exceptions
    end transaction;
  exception -- current alternate has failed, try the next one
    when others =>
      if alternate < N then
        alternate:= alternate + 1;
      else
        raise failure;
      end if;
  end;
end loop;
if raise_exception then
  Reraise_Exception(external_exception);
end if;
```

In this implementation of recovery blocks we use Ada features that allow saving exception occurrences to be signalled later (when the acceptance test is passed).

4.4 Client/server Model and Replication

Recovery blocks were originally proposed as a general means for tolerating design faults [Ran75]. The author was deliberately not specific about many particular details, one of which is applying the scheme in the context of distributed systems. In such context any application willing to tolerate design faults is usually interested in tolerating hardware faults. As we discussed before paper [LABK90] proposes several ways in which hardware and software fault-tolerance mechanisms can be used in combination to tolerate effectively faults of any nature. In particular, in order to tolerate hardware faults during the execution of a recovery block the paper proposes to replicate a recovery block and to concurrently execute replicas on different sites: when a site fails the rest of sites can continue execution and provide a correct result in spite of software design faults.

To apply these ideas in the context of distributed systems we have to distribute the server component implemented with software diversity into several sites. The simplest way of employing recovery block technique for implementing a server component diversely is by designing each of its services as a recovery block. This completely hides diversity from the clients, makes it easier to implement recovery points, which have to only deal with the state of the server and facilitates the run time support. We call this approach service diversity as opposed to another approach which is called server diversity in which the entire server it implemented diversely. Unfortunately the run time support for sever diversity becomes very complicated as it becomes very difficult to keep the states of all diversely-implemented servers consistent.

In order to tolerate site failures two replication techniques can be used here: active replication and the primary-backup scheme. Active replication can be used to avoid the delays in the execution of a recovery block when a site crashes (although the recovery blocks by their very nature can suffer from delays when errors caused by design faults are detected and tolerated) because in this case all (replicas) severs execute the same service concurrently. Group communication can be used to ensure that all replicas receive the same requests in the same order (total order multicast). If there are no hardware faults then, due to replica determinism and total order multicast, all replicas will produce the same results. Either all fail to ensure the acceptance test or ensure it. If due to hardware faults some of the replicas do not complete their execution of the alternate, the remaining replicas proceed with checking the acceptance test and either report the result or rollback and try the next alternate.

In contrast to this approach in the primary-backup scheme only one replica (the *primary*) is running; it periodically multicasts a checkpoint to the rest of replicas (the *backup* replicas). When the primary crashes, one of the backups takes over and continues the execution of the same alternate from the last checkpoint. Therefore, if the primary crashes, some part of the work could be lost and system execution can experience some delays.

Both replication techniques have to deal with the situation when a running replica is allowed to

invoke other servers. If such invocations are allowed, in the case of active replication, they should be filtered so that only one invocation is performed. Special care must be taken in the primary-backup approach to inform backups about the success or failure of such invocations.

The *distributed recovery block* (DRB) [KW89] scheme was the first attempt to tolerate hardware and software faults uniformly. The DRB can be considered a primary backup scheme in the sense that only one site provides results (the primary). However, the other sites execute concurrently different alternates of a recovery block. If the primary fails (crashes or does not pass the acceptance test), the results are provided by a backup. Therefore, the primary must inform the backups whether it has passed the acceptance test or not. In a DRB if the number of alternates of the recovery block is smaller than the number of sites, some sites execute concurrently the same alternate.

In this paper we follow the active replication approach to implement recovery blocks that tolerate site failures because of its simplicity. The programming of a DRB is not straightforward. It can be argued that the DRB can provide results immediately when an alternate fails, however some coordination is needed to let know the rest of the sites about the results of the primary.

4.5 Implementation of Replicated Recovery Blocks

Replication can add the desired level of (site) hardware fault tolerance to recovery blocks. A replicated group can be used to mask site crashes during the execution of a recovery block.

In the code we present there are two services (`Service1` and `Service2`). Each service encapsulates a recovery block, which is programmed as shown in Section 4.3. Group services and exceptions signalled (`failure`, `exception1` and `exception2`) by the group are declared in the group specification. Since the `failure` exception indicates that the execution of a recovery block has failed, it must be always included in the group specification.

```
replicated group specification Recovery_Block is
  -- exceptions signalled by the group
  failure, exception1, exception2 : exception;
  -- group services
  entry Service1(parameters1);
  entry Service2(parameters2);
end Recovery_Block;
```

All group members are exact replicas and implement those two services. Replicas must be executed at different sites to tolerate site crashes. The programmer writes the code for one replica. The server is configured to run a number of replicas at different locations.

```
for group Recovery_Block;
agent Agent1 is
-- data declaration
begin
  select
    accept Service1 (parameters1) do
      -- recovery block for service1
      ok:= false;
      alternate:= 1;
      while alternate <= N and not ok loop
        begin
          begin transaction
            case alternate of
              1: alternate1(params);
              2: alternate2(params);
              ...
            end case;
            ok:= acceptance_test(params);
            if not ok then
              raise failure;
            end if;
          end transaction;
        exception
          when others =>
            if alternate < N then
              alternate:= alternate + 1;
            else
              raise failure;
            end if;
        end;
      end loop;
    end Service1;
  or
    accept Service2 (parameters1) do
      -- recovery block for Service2
    end Service2;
  end select;
end Agent1;
```

Each `accept` statement contains the code of a service, which is programmed as a recovery block. Replication is transparent to clients. Therefore, they invoke services as if they were not replicated. As far as a primary component is available, the service (recovery block) will be executed. The results are filtered and those of one replica are sent to the client, no matter if it is an exception or regular results.

5 N-Version Programming

5.1 The Basic Mechanism

N-version programming (NVP) [Avi85] is another technique that also uses redundancy to mask software faults. In this approach, N versions of a program (a module) are developed independently by different programmers, to be run concurrently. Their results are compared by an adjudicator. The simplest way is to use the majority voting here: the results produced by the majority of versions are

assumed to be correct, the rest of the versions are assumed to have errors, their faults having been triggered in the execution. This technique requires a special support to control the execution of versions and the adjudicator and pass the information among them. In particular, it synchronizes version execution to obtain information (e.g. results) from all of them to pass to the adjudicator. The functionalities of the controller in the first experiments with NVP were executed by and hidden in a special run-time support, DEDIX [AGK+85].

5.2 Client/server Model and Groups of Versions

In order to tolerate site crashes in distributed systems each version of an N-version programming can be executed at a different site: if any site crashes the remaining sites can continue their execution and provide the required service. For the simple majority voting this means that this scheme is capable of tolerating K design faults or site crashes where $K < N/2$.

We believe that server diversity (see Section 4.4) should be applied in the client/sever model for implementing N-version programming because it allows the entire sever components to be designed diversely and independently encapsulating all internal data (employing service diversity would restrict programmers in design services as all implementations of a service have to use the same component data).

It seems very reasonable to use a group of processes to introduce N-version programming into distributed systems. Each group member is a version of an N-version program. That is, processes execute the same set of services but they have different code [LCN90]. For N-version programming we build such a group out of N versions of a diversely-designed server, so that each group member executes a corresponding version. One member, for instance the one with the smallest identifier, can act as group coordinator to be in charge of voting and returning the results to the client. Another approach to executing the voting is by returning all version results directly to the client and performing voting at the client side. Although the former solution hides the voting function from the client, the code becomes complex because it needs to take into account coordinator crashes. In the following section we will demonstrate how the later approach can be implemented.

Recovering faulty versions is known to be a very difficult problem because of diversity of the internal version states. Several techniques have been developed; for example, the community error recovery [TA87] mainly relies on the assumption that all versions have a common set of internal data, whereas approach in [Rom00b] is based on developing an abstract version state and mapping functions from this state to a concrete state of each version and back. The same techniques can be used for recovering versions after site crashes in general, and for implementing a recovery service for processes used for N-version programming, in particular. For example, if all the group members have the same state the state transfer can be performed automatically. However, in a more general case when group members have different states the abstract version state will be transferred among the group members and mapped into a concrete state of the version to be recovered. This function must be called immediately after the recovered site joins the group and before starting processing group request.

5.3 Implementation of N-version Programming using Cooperative Groups

Cooperative groups in *Transactional Drago* have all the functionality needed to implement N-version programming. A cooperative group has a specification describing the services the group provides. Group members are processes that implement the group specification. Clients invoke services using multicast: each request is sent to all members. Members execute the request in parallel. When a member finishes, it returns its results to the client. The client is blocked until all the available members have replied. The number of group members is given by the number of versions. A voting function must be called at the client side immediately after each service of a diversely-designed server is called.

A sample group specification in *Transactional Drago* for an N-version group is the following:

```
group specification N_Version is
  -- group services
  entry Service1(parameters1);
  entry Service2(parameters2);
end N_Version;
```

A group member must implement those two services.

```
for group N_Version;
agent Version1 is
-- group member data declaration
begin
  select
    accept Service1(parameters1) do
    -- version1 for service Service1
    end Service1;
  or
    accept Service2 (parameters1) do
      -- version1 for service Service2
    end Service2;
  end select;
end Version1;
```

At the client group services are called as a regular procedure, *Transactional Drago* multicasts the invocation to all group members and collects all the answers.

```
-- client code
Service1(parameters1, results);
Voting(results, resultsService1);
```

The `results` parameter is an array with all the group members results. Each array component contains the results of a group member. The `Voting` procedure is in charge of returning the majority result.

5.4 N-version Programming and Exception Handling

Allowing servers to signal external exceptions to the clients is an important feature that supports recursive system structuring and promotes disciplined approaches to tolerating faults of many types.

Such exceptions are used in all situations when a server cannot deliver the required results (for example: illegal input parameters, environmental faults, delivering partial results, server failure). Paper [Rom00a] proposes a scheme that allows diversely-designed servers to have external exceptions in their interfaces. Clearly all of the versions have to have the same set of external exceptions because such exceptions have to be treated as an immanent part of server specification. The scheme requires an adjudicator of a special kind to allow interface exception signalling when a majority of versions have signalled the same exception.

This feature can be introduced into the distributed client/server setting when N versions of a server are structured as a group to facilitate the N-version programming support and to allow tolerating hardware faults (Section 5.2). The only extension which is needed is an ability to pass an identifier of an exception that a group member has signalled to the group coordinator (it can be either one of the group members or the client itself) and an extended majority voting that adjudicates not only normal results but exception identifiers as well.

5.5 Implementation of N-version Programming with Exceptions

We have already shown (Section 4.5) that groups can declare external exceptions in the group specification. Those exceptions are propagated to the client when are not handled in the group during the execution of a service. Cooperative group members might have different code and therefore, each member can signal a different exception for the same service. When this situation happens, by default, the predefined exception `several_exceptions` is propagated to the client. However, this behaviour can be overwritten. The group programmer can define an exception resolution function in the group specification for each group service. This function is invoked by the run-time when two or more group members finish the execution of that service signalling different exceptions. This function can be programmed so that the majority exception is signalled.

```
group specification N_Version is
  exception1, exception2 : exception;  -- exceptions signalled by the group
  -- group services
  entry Service1(parameters1);
  entry Service2(parameters2);
private
  use Ada.Exceptions;
  -- predefined type Exception_ID_List_Type is
  --      array (Positive <>) of Exception_ID;
  function MajorityResolution
     (exceptionList: Exception_ID_List_Type): Exception_ID is
  begin
    -- select the majority exception
  end Service1Resolution;
  for Service1'resolution use MajorityResolution;
  for Service2'resolution use MajorityResolution;
end N_Version;
```

The `MajorityResolution` function selects the majority external exception. This function is associated to both `Service1` and `Service2` by means of the clause:

for `Service1'resolution` **use**.

6 Discussion

The idea of using transactions as a means for rolling a system state back is briefly mentioned in [RCS93]. The authors show how the *Arjuna* transactions [SDP91] can be used to support state restoration while employing recovery blocks. However, the paper does not address the problems of alternate distribution and site crashes.

The solutions put forward in our paper use *Transactional Drago*; they could, however, be implemented using any library or toolkit that supports group communication (e.g., *Ensemble* [Hay98], *Totem* [MMSA$^+$96] or *Transis* [DM96]) and transactions (e.g., *Arjuna* [SDP91] or *TransLib* [JPAB00]). These implementations would be more flexible and efficient (for instance, locks are only set when they are needed) but more error prone (because, for example, the programmer has to state explicitly which data are recoverable and which locks are needed).

The Fault Tolerant CORBA [OMG00] provides support for groups of objects which, in particular, includes an active replication style. This style corresponds to *Transactional Drago* replicated groups, with each service corresponding to a method. CORBA active replication and *Transactional Drago* replicated groups have the same chief properties: replication transparency, determinism and total order. In the Fault Tolerant CORBA *active_with_voting* replication style both requests addressed to and replies from a replicated group are voted and majority results are delivered. This style can be used for implementing N-version programming; it is not, however, supported in the current specification of the Fault Tolerant CORBA. This specification supports automatic replica recovery by replaying a log file to bring a new replica to a consistent state. Since CORBA also supports transactions, both recovery blocks and N-version programming can be programmed in ways similar to those demonstrated in the paper.

7 Conclusions

Although techniques for software (design) fault tolerance have been studied for a very long time, there are not many programming languages or toolkits that incorporate them. Programmers do not get much support when they are faced with the task of applying N-version programming or recovery blocks. By the contrast, software techniques for tolerating site failures, namely transactions and replication (built on top of group communication), have become very popular and are part of most middleware platforms used for developing distributed applications. In this paper we have shown how recovery blocks and N-version programming can be programmed using transactions and group communication features as building blocks. The proposed techniques allow site failures and design faults to be treated uniformly. In addition, the paper discusses how recovery blocks and N-version programming can be extended to incorporate disciplined exception handling and how such extended schemes can be implemented using groups and transactions. The proposed approaches are demonstrated in *Transactional Drago*, an Ada extension for fault tolerant distributed programming, but they can be easily programmed using other toolkits supporting transactions and group communication. For example, CORBA and the fault tolerant CORBA provide all features required for implementing the proposed schemes.

References

[Ada95] *Ada 95 Reference Manual, ISO/8652-1995*. Intermetrics, 1995.

[AGK+85] A. Avizienis, P. Gunningberg, J. P. J. Kelly, L. Strigini, P. J. Traverse, K. S. Tso, and U. Voges. The UCLA DeDiX System: A Distributed Testbed for Multiple-Version Software. In *Proc. 15th Int. Symp. on Fault-Tolerant Computing (FTCS-15)*, pages 126–134, Los Angeles, CA, 1985. IEEE Computer Society Press.

[AOS+99] K. Arnold, B. O'Sullivan, R. Sheifler, J. Waldo, and A. Wollrath. *The JINI Specification*. Addison Wesley, 1999.

[Avi85] A. Avizienis. The n-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, 11(12):1491–1501, 1985.

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, Reading, MA, 1987.

[CKV01] G. V. Chockler, I. Keidar, and R. Vitenberg. Group Communication Specifications: A Comprehensive Study. *ACM Computer Surveys*, 33(4):427–469, December 2001.

[Cri89] F. Cristian. Exception handling. In T. Anderson, editor, *Dependability of Resilient Computers*, pages 68–97. BSP Professional Books, 1989.

[DM96] D. Dolev and D. Malki. The Transis Approach to High Availability Cluster Communication. *Communications of the ACM*, 39(4):64–70, 1996.

[FvR95] R. Friedman and R. van Renesse. Strong and Weak Virtual Synchrony in Horus. Technical Report TR95-1537, CS Dep., Cornell Univ., 1995.

[GMAA97] F. Guerra, J. Miranda, A. Alvarez, and S. Arévalo. An Ada Library to Program Fault-Tolerant Distributed Applications. In K. Hardy and J. Briggs, editors, *Proc. of Int. Conf. on Reliable Software Technologies*, volume LNCS 1251, pages 230–243, London, United Kingdom, June 1997. Springer.

[Hay98] M. Hayden. The Ensemble System. Technical Report TR-98-1662, Department of Computer Science. Cornell University, January 1998.

[Jal94] P. Jalote. *Fault Tolerance in Distributed Systems*. Prentice Hall, NJ, 1994.

[JPAB00] R. Jiménez-Peris, M. Patiño-Martínez, S. Arévalo, and F.J. Ballesteros. TransLib: An Ada 95 Object Oriented Framework for Building Dependable Applications. *Int. Journal of Computer Systems: Science & Engineering*, 15(1):113–125, January 2000.

[KW89] K. H. Kim and H. O. Welch. Distributed Execution of Recovery Blocks: An Approach for Uniform Treatment of Hardware and Software Faults in Real-Time Applications. *IEEE Transactions on Computer Systems*, 38(5):626–636, May 1989.

[LABK90] J. Laprie, J. Arlat, C. Béounes, and K. Kanoun. Definition and Analysis of Hardware- and Software-Fault-Tolerant Architectures. *IEEE Computer*, 23(7):39–51, 1990.

[LCN90] L. Liang, S. T. Chanson, and G. W. Neufeld. Process Groups and Group Communications. *IEEE Computer*, 23(2):56–66, February 1990.

[MMSA+96] L.E. Moser, P.M. Melliar-Smith, D.A. Agarwal, R.K. Budhia, and C.A. Lingley-Papadopoulos. Totem: A Fault-Tolerant Multicast Group Communication System. *Communications of the ACM*, 39(4):54–63, April 1996.

[Mos85] J. E. B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. MIT Press, Cambridge, MA, 1985.

[MSR77] P.M. Melliar-Smith and B. Randell. Software Reliability: the Role of Programmed Exception Handling. In *In Proc. of an ACM Conf. on Language Design for reliable Software*, pages 95–100, 1977.

[OMG] OMG. *CORBA services: Common Object Services Specification*. OMG.

[OMG00] OMG. *Fault Tolerant CORBA*. Object Management Group, 2000.

[PJA98] M. Patiño-Martínez, R. Jiménez-Peris, and S. Arévalo. Integrating Groups and Transactions: A Fault-Tolerant Extension of Ada. In L. Asplund, editor, *Proc. of Int. Conf. on Reliable Software Technologies*, volume LNCS 1411, pages 78–89, Uppsala, Sweden, June 1998. Springer.

[PJA01] M. Patiño-Martínez, R. Jiménez-Peris, and S. Arévalo. Exception Handling in Transactional Object Groups. In *Advances in Exception Handling, LNCS-2022*, pages 165–180. Springer, 2001.

[PJA02] M. Patiño-Martínez, R. Jiménez-Peris, and S. Arévalo. Group Transactions: An Integrated Approach to Transactions and Group Communication. In *Concurrency in Dependable Computing*, pages 253–272. Kluwer, 2002.

[PJKA02] M. Patiño-Martínez, R. Jiménez-Peris, J. Kienzle, and S. Arévalo. Concurrency Control in Transactional Drago. In *Proc. of Int. Conf. on Reliable Software Technologies (in publication)*, Vienna, Austria, June 2002. Springer.

[Ran75] B. Randell. System Structure for Software Fault Tolerance. *IEEE Transactions on Software Engineering*, 1(2):220–232, 1975.

[RCS93] C. M. F. Rubira-Calsavara and R. J. Stroud. Forward and Backward Error Recovery in C++. Technical Report Technical Report 417, Dept. of Computing Science, University of Newcastle upon Tyne, 1993.

[Rom00a] A. Romanovsky. An Exception Handling Framework for N-Version programming in Object-Oriented Systems. In *Proc. of the 3rd IEEE Int. Symp. on Object-oriented Real-time Distributed Computing (ISORC'2000)*, pages 226–233, Newport Beach, USA, March 2000. IEEE Computer Society Press.

[Rom00b] A. Romanovsky. Faulty Version Recovery in Object-Oriented Programming. *IEE Proceedings - Software*, 147(3):81–90, 2000.

[Sch90] F. B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.

[SDP91] S. K. Shrivastava, G. N. Dixon, and G. D. Parrington. An Overview of Arjuna: A Programming System for Reliable Distributed Computing. *IEEE Software*, 8(1):63–73, January 1991.

[Sun] Sun. *Enterprise JavaBeans*. http://java.sun.com/products/ejb/index.html.

[TA87] K.S. Tso and A. Avizienis. Community error recovery in N-version software: a design study with experimentation. In *Proc. of 17th IEEE FTCS*, pages 127–133, Pittsburgh, PA, 1987.

[XRR98] J. Xu, A. Romanovsky, and B. Randell. Coordinated Exception Handling in Distributed Object Systems: from Model to System Implementation. In *Proc. of the 18th IEEE Conf. on Distributed Computing Systems (ICDCS18)*, pages 12–21, Amsterdam, The Netherlands, May 1998.