Flexible Schemes for Application-Level Fault Tolerance

L. Strigini, F. Di Giandomenico
IEI-CNR, Pisa

Abstract

This report considers application-level fault tolerance, that is, fault tolerance policies explicitly programmed in the upper layers of a (hardware plus software) computing system. Design of fault tolerance at this level is normally necessary, but difficult and error-prone due to its mostly ad-hoc nature. There is a need for discipline to control the complexity added to any design by the provision of fault tolerance, and to allow the automation of some part of the task of redundant design. At the same time, the structuring schemes proposed in the literature appear too restrictive for the building of large, heterogeneous applications. Ways to overcome these limitations while preserving a structuring discipline are discussed first with respect to the detection and signalling of errors, and then, to a wider extent, to recovery.
Flexible schemes for Application-Level Fault Tolerance

Lorenzo Strigini, Felicita Di Giandomenico

IEI - CNR, Pisa

Abstract

This paper discusses some problems with application-level fault tolerance, that is, fault tolerance policies explicitly programmed in the upper layers of a (hardware plus software) computing system. Design of fault tolerance at this level is normally necessary, but difficult and error-prone due to its mostly ad-hoc nature. There is a need for discipline to control the complexity added to any design by the provision of fault tolerance, and to allow the automation of some part of the task of redundant design. At the same time, the structuring schemes proposed in the literature appear too restrictive for the building of large, heterogeneous applications.

We discuss the signalling of errors between components and the coordination of recovery. There is a need to allow the combination of components employing different basic schemes for recovery, because the schemes that can be used in the individual components of a system are usually dictated by requirements or inherent characteristics of the component themselves. To avoid excessive complexity, it would be useful to state rules about the combination of components that employ different schemes. These would specify constraints for the designers of the components to be interfaced, and requirements on the virtual machine supporting their execution. As an example, we propose a solution for interfacing subsystems using conversations for backward recovery with subsystems using planned atomic transactions. We discuss how a classification of components could be organized to allow the formulation of more general solutions.
Introduction

As fault tolerance becomes a requirement in wider fields of application of computers, there is some difficulty in extending the simple schemes that have proven successful in specialized applications. In particular, large application systems combine subsystems differing in style of organization, error modes and dependability requirements; if built on distributed systems, they introduce new design problems, like coping with parallelism of execution, variable delays, and distributed consistency.

Well-structured design of computing systems is usually based on regarding the system as a stack of virtual machines. Each machine is built using the services of the lower-level one[s], and provides a set of services that are more convenient, in some way, for its user (machine, designer, human user), either complementing or hiding the machine under it (virtual machine, [run-time] support, interpreter). Components in the higher layers are called application components. In the design of fault tolerant computing systems, measures for detecting and tolerating faults can be applied at all levels. Applying fault tolerance in the lower levels is useful in that it frees the designers of the higher levels from consideration of faults in the lower levels, shares the cost of fault-tolerant design among all the applications that use the same lower-level machine, and allows the designer to employ comparatively cheap techniques to cover known frequent faults (e. g., memory faults). On the other hand, the tolerance of faults cannot in general be completely left to the lower levels in a machine, because:

- errors occur that do not violate the constraints of the underlying virtual machines. Since lower-level error detection mechanisms are based on such constraints (e. g., "the contents of a memory word must always have even parity"), they cannot detect such errors. Such undetected errors may originate in physical faults or in the implementation of the higher levels. On the other hand, the application often has ways of detecting them, based on known properties of the application data (e. g., bounds on the values of a program variable representing a physical variable; properties of application-specific data structures). A typical example is that of errors caused by bugs in the application software itself: one of the motivations for application-level fault-tolerance is the need to tolerate design faults by robust design of the software;

- recovery, from any kind of errors, if planned at a lower level, in an application-independent fashion, could be ineffective (e. g., roll-back and retry in the presence of a software bug may well not correct the errors; even if it did, it may violate some real-time requirement). Instead, the application programmer has knowledge about the goals and constraints of the application that is not available to the designer of the underlying virtual machine. So, the former can program application-specific recovery actions like forward recovery of data bases, compensation of external actions, reorientation of the computation activity towards back-up goals.

So, fault tolerance must also be a consideration in the design of the higher levels, and actually it may be more cost-effective to save on low-level mechanisms and put more effort into fault tolerance in the higher levels, to avoid duplication of efforts (this is often called the end-
to-end argument, after [Salzer 1981]). This argument applies throughout the design of a complete system, up to the application level.

Nowadays, standard techniques exist for applying fault tolerance in the hardware and executive layers of systems for such problems as masking hardware faults, without reconfiguration, by strictly synchronized N-modal redundancy. error correcting codes [Carter 1985] and such; or organizing the recovery of application processes in a distributed environment to tolerate clean hardware faults: standard solutions are implemented in commercial machines (Tandem [Dimmer 1985]), and widely available packages (ISIS [Birman 1990]).

Instead, application-level treatment of generic faults and errors is a complicated problem for any designer. There are obviously no application-independent solutions: help may come from the use of some general structuring scheme for managing the complexity of redundant application-level design, and some set of feasible lower-level mechanisms that would ease the application of such schemes. There are studies in this direction, done in the last fifteen years or so, mostly under the headings of "exception handling" (one way of structuring the treatment of abnormal situations) and "software fault tolerance" (programming to tolerate the effects of some design flaws). This latter body of literature contains several proposals for structuring application-level fault-tolerance in concurrent or distributed applications. These can be roughly categorized into:

- redundancy based on system-supported backward recovery with mechanisms for controlling the propagation of roll-back after errors: recovery blocks, conversations and such;
- modular redundancy with masking, as in multiple version programming;
- atomic transactions, using application-level, component-specific backward recovery;
- forward recovery based on some structured form of exception handling;
- inherently fault-tolerant algorithms;
- use of audit programs on redundant data bases (this last method cannot be considered as a self-sufficient structuring method; the reason why we list it separately is that it creates some peculiar problems).

Simple, "pure", structuring schemes are certainly applicable in the design of small, homogeneous simple systems [Anderson 1983, Brunette 1983, Kopetz 1989]. But they seem to disallow useful (or, at least, commonly used) practices, both in the structuring of the "functional" part of applications and in the provision of redundancy; applying such schemes in large systems designed with current programming languages is thus prima facie impossible [Gopal 1990, Gregory 1989]. In practice, each proposal of a scheme for organizing fault tolerance is normally presented within a scenario that includes the use of a specific design style (corresponding to a simplified computational model, e. g. client-server, coroutines, free message passing processes, transactions on long-lived, passive data, data-flow), besides the specific style of fault tolerance that is proposed. Little attention is generally paid to exporting
the proposed fault tolerance concepts to another style of design, or applying them to systems employing mixed styles, as may be required in large real systems\footnote{A question is whether, in real systems, we actually need mixed styles in the same interpretation level. One could maintain that, as a matter of simplicity, they should always be kept separate, as, e.g., when employing coroutine-style consensus protocols under client-server style application. At present, complete purity is impractical: for instance, a pure message-passing organization without shared data among coresident processes is inefficient on most current architectures.}.

Likewise, contributions in the literature are mostly concerned with studying an individual scheme for fault tolerance, while we may expect that in any large system, different kinds of error detection and recovery mechanisms will be suitable for different parts of the system [Strigini 1990, Strigini 1985]. The choice of one or another style of redundancy will be to a large extent dictated by the part of the application where it is to be applied. For instance, regarding error detection, the merits of using an acceptance test depend on an evaluation of whether testing for those properties of the result that can be efficiently checked is deemed to provide sufficient coverage, and whether it is acceptable to defer detection until a stage of the computation where the test can be applied; comparison of the results of diverse variants of a component would be attractive if suitably diverse algorithms are known for performing the intended function; and so on. The costs of the different approaches also depend on the function of the individual component under consideration.

Even projects that have built operating system mechanisms for supporting software fault tolerance [Avizienis 1985, Purtilo 1989] have not paid much attention to the need for combining different schemes of software fault tolerance (in the same computer and possibly in the same application). Although it is easy, when designing “in the small”, to allow blocks or procedures in sequential programs to be implemented in a multiple-version or recovery block technique, the coordination of sets of concurrent processes that employ these techniques could be realized in different ways, with different properties and requirements on the individual components and their interpreter.

Summarizing, we can state that

- there are several basic philosophies for structuring fault tolerance, each one usable with a certain specific design philosophy and requiring specialized support; for each basic fault tolerance philosophy, there are several possible implementations, differing in details;

- although the individual schemes appear useful and sound, there is insufficient experience to show how easy they would be to use in real applications (let alone how well they would combine together), and in particular which of several proposed schemes based on the same basic concepts (for instance, conversations [Horning 1974, Randell 1975] as implemented in [Kim 1982] and colloquies [Gregory 1985]) should be considered the best in practice (possibly as a function of the individual constraints of each particular application environment);
it appears that real applications would contain parts requiring the use of the basic ideas of several of these schemes, jointly or separately;

implementing the specific support mechanisms required by any one of these schemes is feasible, but is certainly a serious investment;

the problem of "interoperability" of these schemes together, and compatibility with existing design styles and programming languages, should be considered before choosing any one scheme for implementing its basic support mechanisms in the virtual machine under consideration.

In this paper, we concentrate on unifying schemes, and in particular on how different styles of redundancy, natural for different application components, can be combined in the same system.

Most of the complexity in redundant software arises from the difficulty of visualizing the possible interactions among redundant modules taking into account both "normal" and "exceptional" modes of operation. It is necessary to seek styles of design that limit the complexity of the organization of fault tolerance. We shall limit ourselves to a level of aggregation in a design where source code is not visible, and the system is represented as a set of components of few (possibly one) basic type with well-defined, restricted communication primitives (each component may in its turn be seen as a system, further decomposable to the degree that is considered useful). The selection of such a style is equivalent to the selection of a system description (or "programming in the large", or "module interconnection") language. Two problems exist: selecting a language that facilitates the design of fault tolerance policies (with the economic constraints for each style, of not requiring too expensive support mechanisms and satisfying a wide enough class of applications), and then selecting standard schemes or design templates for fault tolerant structures, described in that language, that can be applied to the design of many components.

Another important practical issue is that of separation of concerns and information hiding: the use of information about the internal design of a module should only be required of the designer of that module. Without such limits, the maintenance and extension of a software system, and the reuse of software components, can become unmanageable. In particular, the principle must be extended to offer the ability to ignore, in designing a component, exactly what kind of redundancy is provided by other, interacting components: the fault tolerance policies must only use operations made available at the interfaces of the other components via standard communication primitives. The overall design of redundancy and evaluation of dependability of a system must of course take into account the redundancy available in all different modules, but this should not add to the burden of individual component designers. It is also useful to be able to ignore, at some level of detail, as much as possible of the redundancy inside the individual modules, while seeing only the redundancy that counts at this level of detail. Besides, it is often desirable to have a design where fault tolerance, for the whole design, can be examined separately from the part of the design that implements the normal (fault-free) behaviour. This is achieved, for instance, in exception-handling notations by separating the code for fault treatment from the code for fault-free execution, in the same
level of detail, and in multiple modular redundancy by having a simple mapping of a simple
design onto the N-modular redundant design.

The rest of the paper is organized as follows. Section 1 considers the structuring of
error detection and signalling between components, discussing cases where no clear hierarchy
exists along which error reporting can be organized. In Section 2 we consider recovery; as an
example, we concentrate on backward recovery and in particular on the conversation scheme
[Randell 1975], with its proposed implementations and extensions. We argue that extensions
are needed to allow the combination of components designed along different schemes: Section
3 proposes an example, namely the combination of conversations with atomic transactions, and
Section 4 briefly discusses generalizations from that example.

1 Error detection and signalling between components

1.1. The problem; idealised fault tolerant components

A preliminary consideration for fault tolerance is usually how errors can be detected and
how this detection can trigger the proper reaction. The means for detection are various, at all
levels in a system, and individually well understood. We consider here the problem of
signalling the detection of an error to the component[s] that are designed to try and correct it:
the "glue" for the system-wide organization of fault-tolerance.

With effective error confinement mechanisms, it may be unnecessary to propagate error
signals among components. In general, however, the detection of an error is used to prompt
activity meant not only to confine its spread, but to recover from its effects (that may have
spread outside the originating component) as well. A further problem arises from the fact that
information needed for acceptance tests on the results of a component is often contained inside
another component that uses those results (while the producer may have the means for
recovering from an error). Consider for instance a software component S that reads a sensor,
computes from its reading the value of some physical variable, and sends it to other
components. A reasonableness test on the measurement may be based on properties of the
sensor and the variable itself (e.g. continuity, maximum rate of variation), and thus be
programmed in S itself. But it may also be based on properties of the situation that the variable
describes, which is typically known to some other component. For instance, in an airplane, if
the airspeed is zero, then a reading of zero weight on the undercarriage is probably erroneous;
the integration of the two data items is done in a component different from those that produce
them. It is then necessary to let S know of the inconsistency detected by some other
component, or to give S additional knowledge to perform the additional checks. The latter
solution would, if carried out throughout a system, defeat information hiding and lead to a
great confusion of interfaces. It may also happen that the detection of an error makes
untrustworthy data that have already been propagated to other components, so requiring a
propagation and coordination of the recovery activity.

For sequential programs, exception handling has long been studied, to integrate error
signals from all sources into a single recovery invocation mechanism, easy to organize to match
the proper action to each error situation. [Anderson 1981], with the *idealised fault-tolerant
component*, generalizes the concept to any client-server structuring (or, at a conceptual level
only, interpretation hierarchy). The interfaces among components are enriched with the ability to send exceptional replies, allowing the signalling of an error from the component that detected it to a component that is immediately higher in the call hierarchy. This scheme preserves the separation of concerns and information between communicating components, while providing for the propagation of the knowledge of errors between components. One kind of exceptional reply signals an interface exception: the server tells the client that its request is unacceptable. So, if the server finds unreasonable values of the parameters of a call, it can signal an interface exception; if it is the client that is not satisfied with the reply of the server, it can repeat its call (with possible compensatory calls for the effects of the previous one), although it cannot usually directly signal a request for recovery (an undo operation may be allowed by the interface of the individual server, but is not required as a standard interface operation). To control and coordinate recovery, operations on servers may be made atomic and possibly grouped in transactions.

Another extension is proposed in [Campbell 1986] through the use of exception resolution trees. For a group of components that belong to a same "atomic action", all exceptions are propagated to all components, and a discipline is described (by the tree) to decide which handlers will be invoked in each component. This should allow the designer to group in the same atomic action state changes with the checks on which their commit must depend, even if performed in parallel in different components.

1.2. Back-propagation of error detection in message passing

A version of the idealised fault-tolerant component concept for systems built on a general message-passing structure could use an expanded send/receive construct, as in Fig. 1.

receiver:
receive (parameters) subject to predicate
  on time-out action1
  on failure action2

sender:
send (parameters) on refusal action

Fig. 1a Syntax for recoverable communication with acceptance test at the destination

The receiver component can test the contents of the received message before accepting it: if the contents of the message fail to satisfy predicate, then the interpreter will cause the sender to execute the on refusal part of the send construct. The action taken by the sender to recover from the perceived error may be of any kind, including executing a diverse variant of the same computation (recovery block). When the sender is unable to provide a (farther) alternative input, the receiver executes the on failure part of its code, so attempting to recover locally. The separate on time-out statement allows the programmer of the receiver component to plan for the case where the sender component does not deliver the data by some set deadline (timing error), separately from the case where the sender component only provides unacceptable data (value error).
Similarly to the idealised fault tolerant component, this construct allows the structure of a system to be the same as though no recovery were planned, and yet exploit whatever recovery capabilities are independently provided in the two communicating components; the designer of a component need not know the degree of redundancy available in the other module. The inability to back-propagate error detection would cause what [Kim 1986] calls (referring to programmer-transparent recovery) the *exhausted importer* problem: an imperfect acceptance test by a sender may cause a receiver to fail its own acceptance test, and exhaust its local error recovery abilities to no avail, since the only practical solution would be to substitute the erroneous input with a correct one. As with the fault-tolerant recoverable component, preserving consistency through the propagation of recovery is a separate concern: it is the programmer's responsibility to keep track of the past operations (including communications) that must be undone or compensated when a send is unsuccessful.

If a recoverable send operation is allowed inside the evaluation of the predicate in a recoverable receive, then the back-propagation of error signals can proceed through several stages in a sender-receiver chain. This of course creates both opportunities and problems. However, the depth of propagation can be controlled by knowing which components in the chain have a recoverable send within their recoverable receive.

1.3. Audit programs (and signalling of exceptions between application components)

A last, most difficult case is when no explicit communication event is programmed between the component that detects the error and the one that has to react to it. This is the case with errors detected at a level of interpretation that is not visible to the application programmer (e. g., hardware traps), which is naturally dealt with as an extension to exception handling mechanisms. Another example is given by audit programs.

Audit programs are programs that examine the status of a [sub]system in order to detect, and possibly correct, errors. The audit program must be able to access the information belonging to the application, check that certain "consistency predicates" are satisfied, and possibly correct the data items that are likely to be erroneous. This creates problems in several areas:

- a conflict with the principle of information hiding in the design phase [Gopal 1990]: the programmer of the audit program must know the intended meaning of the data stored in the data structures to be "audited", and possibly also their physical organization in storage;

---

2 Substituting this communication primitive for synchronous message-passing, without changes to the communicating modules, would not affect the correctness of a system (in the absence of faults). With asynchronous messages, the additional synchronization introduced between communicating modules by allowing back-propagation would have to be taken into account in designing the system.
• protection of data at run-time. The audit program should be able to access data that are otherwise private to other components. If the consistency predicates involve data belonging to several such components, the access privileges of the audit program might become exceedingly wide, and thus dangerous;

• concurrency control. In the frequent case where auditing is performed as a background task, there is the risk that the audit program (in particular if it has very wide privileges) can see intermediate inconsistent states of data while the audited program is performing some action on the data;

• notation and communication with the audited component. If recovery is the responsibility of the audited applications, the audit program must be able to inform them about the problems detected. A reasonable way could be the raising of exceptions in the audited program, but to this purpose the audit program must have some privileged, protected access to the interpreter.

All these problems are greatly reduced if auditing is done via explicit enquiries by the audit program to the audited program, instead of by sharing of variables. If the audited program is a server process encapsulating some data item, it becomes quite easy for it to hide the detailed physical organization of the data, to enforce the appropriate concurrency control policy, and to perform some handshake to obtain either a "certificate of compliance" or directions for corrections. Of course, this organization may create problems of efficiency, since it may require the transfer of much information; it trusts the audited program for what concerns the mapping between the physical and logical organizations of data, and the concurrency control policy, so that it may fail to detect some errors; and it requires that the audited program be organized in such a way to allow these queries. If these constraints are unacceptable, it becomes necessary that the audit program effectively "spies" upon the data space of the audited program[s]. This may be made less dangerous by a flexible protection scheme enforced in the interpreter, designating different access rights for different areas: so, the audit program might be prevented from overwriting information that it has no business overwriting, or even compelled to respect locks set by the application, thus freeing the programmer of the audit program from problems of concurrency control. A problem with this approach is that flexible, fine-grain protection mechanisms are not widespread in current architectures; another problem might be in the fact that the protection directives for the data in a program could not be represented in current high-level programming languages, and so they would have to be described in a separate design document, complicating the overall design.

2 Backward error recovery in a message-passing environment - conversations

2.1. Generalities

In this model, the system in execution is composed (Fig. 2aa) of a number of processes that communicate only through messages: no shared variables exist. There is no a priori constraint on the pattern of message passing.
Fig. 2aa A system of message-passing processes. Vertical lines represent processes, arrows represent messages; the vertical axis represents time.

When an error is detected in the state of a process, the process is rolled back to a previously saved state (checkpoint) and a retry is performed. The checkpointing of the state, the detection of errors, and the retry can all be either organized by the underlying virtual machine or programmed in the application software (with support from the virtual machine). A way of organizing them in the application is the recovery block [Randell 1975]: at the entry in the block, a checkpoint is taken; diverse alternates are provided for the code that executes the operations of the block; if the results of the execution of an alternate fail an acceptance test, the process is rolled back to the checkpoint and another alternate is invoked.

In an environment of communicating processes (or generic message-passing components), the roll-back of one process, P, may require the roll-back of other processes that communicated with it. Three solutions are normally considered: isolated backward recovery, programmer-transparent coordination, or programmed coordination. We briefly recall them.

Backward recovery can be confined within a process (or generic component) P if P has the properties of fail-silence (it never emits incorrect messages towards the rest of the system) and determinism (running from a given initial state, and receiving a given sequence of messages, P will always send exactly the same sequence of messages and end up in the same final internal state). Then, it is sufficient that the underlying virtual machine, during the retry, replays to P the messages it received after the checkpoint and before the failure and stops the sending of all messages that P sends during the retry that duplicate messages sent during the original run [Birman 1984, Powell 1988]. A degenerate case (no longer requiring determinism) is obtained if a checkpoint is taken before sending each message.

If the above hypotheses are not both verified, the roll-back of one process, P, may in general require the roll-back of other communicating processes (though application specific cases can be found where this does not apply), both to undo the effects of errors possibly propagated from P and to ensure consistency among the states of P and the other processes. This propagation can easily be automated (chase protocol), but may be excessive and costly, in terms of time, and amount of recovery information to be preserved for system recovery to be possible (domino effect). The domino effect can be controlled by a coordination of checkpoints and roll-back, either programmer-transparent [Barigazzi 1983, Kim 1990, Kim 1986], or
programmed. Transparent solutions guarantee that enough checkpoints are kept for consistent roll-back when needed, and typically allow some degree of tuning for the extension of roll-back allowed and/or the amount of recovery information stored. Their limitation is that it is difficult to establish beforehand which set of processes will be rolled back when any given error situation arises; besides, if the transparent mechanism has the authority to establish checkpoints, there is the problem of ensuring a convenient test of correctness before the checkpoint is taken. Instead, with explicitly programmed coordination of recovery one can plan in advance the extension of roll-back that will happen. So, it is possible to check the respect of timing constraints, and to plan the combination of subsystems that work with backward error recovery together with others that use other schemes. Proposals for programmed coordination of recovery are mostly based on the concept of conversations [Randell 1975].

2.2. Conversations

A conversation (Fig. 3ab) can be described as a multi-process recovery block: when two or more processes agree to enter a conversation, each must have a checkpoint representing its state at the beginning of the conversation, and they may only leave the conversation (committing the results computed during the conversation, and discarding their checkpoints) by consensus that all their acceptance tests are satisfied. During the conversation, they must not communicate with any process outside the conversation itself. Conversations may be nested freely, meaning that any subset of the processes involved in a conversation of nesting level \( i \) may enter a conversation of nesting level \( i + 1 \).

![Fig. 3ab Conversations](image)

Many authors have proposed implementations of the general concept of a conversation, including language constructs and support mechanisms. A detailed discussion of them is in [Di Giandomenico 1991]. Differences among these proposals include:

- how participation in a conversation is determined. In the completely static case, it could be both declared in the code of the involved processes, and known to the run-time support, so that it can be checked that all and only the processes that are intended to take part do actually take part in any conversation. At the other end of the spectrum, a
conversation may be determined dynamically by several processes calling an "enter conversation" primitive, with the same conversation identifier as a parameter.

Dynamic participation in conversations seems to create many problems for the comprehensibility of programs. All processes in a conversation have to synchronize on the exit from the conversation, where all changes made to their states are committed, and this creates an additional synchronization link besides the others intended by the programmer. This may cause delays and even deadlock (which of course is also possible with static participation, but easier to avoid). It is interesting to notice that if a process P inside a (dynamically determined) conversation X sends a message M to another process Q that is not a member of X, the run-time support cannot determine whether Q will ever join X (and the message will be delivered to Q); if the message times out, it is not clear whether this is due to a failure in P (having sent a message to a process that was not meant to be in X), or in Q (failing to join X when intended: Q is a deserter process, in the terminology of [Kim 1982]). Retrying the conversation will not take care of Q's failure. If the other processes in X try to commit while P is waiting on the sending of M, P will appear to be the deserter process. This cannot be cured by making time-outs on commit longer than those on message delivery, since P could enter the conversation an arbitrary time after its partners.

- whether the operations of the participating processes are described separately in the codes of the processes or together in a block of code for the conversation, for instance in a Concurrent Pascal monitor [Kim 1982]. This choice implies a choice of whether a design is decomposed along the dimension of time (a module of decomposition, or unit of responsibility in the design, represents all the activities performed on a set of data during a limited interval of time) or of space only (a module is responsible for all the operations on a limited set of data during a long period of time - it is a long-lived process). The question is relevant for the comprehensibility of the collective operation of the processes involved.

Grouping together all code belonging to a conversation seems somewhat in contradiction with the general assumption of an execution environment made of long-lived processes, which would seem to imply that processes are separately designed. On the other hand, keeping the code separate makes it difficult, for instance, to visualize the nesting of conversations or to program an acceptance test based on some consistency property among data belonging to several processes. A third option is to compel all the processes in the system to resynchronize at each "execution step", as in the concurrent recovery blocks in [Kim 1982] (Fig. 4ad); conversations are restricted to happen among parallel processes spawned (and hidden) inside a phase of operation of a component (with the external appearance of a sequential component). In other words, it is assumed that in practice parallelism is used to efficiently program some limited (in both space and time) part of the application, not to form complex designs. This results in decreased efficiency, and a need to design the whole system as a monolithic entity, a design style that may be acceptable (small, simple systems) or even desirable (e.g. for hard real-time systems, where all steps of computation must be statically scheduled into computation frames), though certainly impractical for large systems with efficiency requirements.
In short, programming with conversations seems to be most natural for an application system with either a short-lived or a cyclical execution, in which each iteration repeats exactly the same static pattern of conversations; as a limiting case, in a single sequential thread with forks and joins, as in Fig. 4ad. These are actually the applications typically designed according to a model with free exchange of messages: simple concurrent algorithms described by a rather static execution and communication pattern, that realize some well-defined task via a regular, predictable pattern of interactions. For these applications, very static implementations of conversations suffice. For instance, [Anderson 1983] uses the assumptions (normal in "hard" real-time systems) that precedence relations among the executions of [subsets of] processes are known in advance, and that the scheduling of communicating processes is restricted to two patterns, to define exchanges, a form of static conversation which is implicitly defined by the design of the system for fault-free situations, not as an additional design concern (exchanges could easily be complemented with back-propagation of errors as described in 2.2). For systems with dynamic scheduling, such precedence relationships can be guaranteed by using the explicit synchronization mechanisms offered by the language along some disciplined pattern; an example is the concurrent recovery block, quoted above. With these static schemes, adding one component to the system requires that the whole sequence of conversations be planned anew, but this seems to be reasonably easy in these two examples.

It is true that many systems cannot be designed in this fashion. When an application requires interaction among components to be initiated by independent decisions of the individual components (as is the case in open or dynamically changing systems), other design methods are necessary. These methods typically control interactions via policies implemented in the individual components that protect data. How this can be integrated with conversation-based recovery is discussed in the next section.

To conclude this section, we mention that the concept of conversation has been criticized as limited under two distinct points of view:

1. the limitations it imposes on the interactions among processes (within the long-lived process, message passing model) and the flexibility of recovery policies; e.g., [Gregory 1985] extends the concept of conversation to allow each retry (alternative) of the recovery block in one process to cooperate with a set of processes different from the previous try.
This should make it easier to survive desertion (by making a time-out on communication within a dialog - implicitly with a deserter - cause the dialog to fail), and in general to allow more diversity among the alternate procedures for reaching the intended goal. Unfortunately, this further complicates the problems implicit in the dynamic creation of conversations. If the retry of a dialog involved a proper subset of the processes involved in the original try, the design would still look manageable (all the recovery activity would be confined within the same modules - conversations or processes). Otherwise, it seems difficult in general to make sure that an alternative partner would be ready to cooperate in a retry at the very moment when it is needed. This style could certainly be applied in particular situations, but it seems hardly worth the building of specific support mechanism and language extensions.

2. the fact that in real-world applications programming styles are necessary that are incompatible with the conversation model. [Gregory 1989] cites shared variables and server processes, and dealing with processes spawned during a conversation.

We have argued that extensions to the conversation scheme meant to allow more dynamic or complex patterns of message passing among processes are of questionable usefulness. Other kinds of refinements seem to be worthwhile, and are discussed in [Di Giandomenico 1991].

3 Conversations with server components

We discuss here the integration of conversations with server components, that is, components that encapsulate data and export operations on those data through a procedure-like interface. Server components are a popular style for programming. They allow clients to decide at run-time to request operations on the data, and protect the consistency of the data by enforcing proper concurrency control policies and application-specific error recovery. Unfortunately, the very dynamicity of client-server interactions seems to forbid the use of server components with conversations, as pointed out for instance in [Gregory 1989].

We discuss here a solution to the problem. A preliminary consideration is that the typical mechanism for structuring backward recovery in server processes or shared data is atomic transactions. All operations on the shared data are divided into groups, called transactions, that transform a consistent state of the object[s] into another consistent state. Each change in the state of the shared object is organized as a recoverable transaction. The clients that request a transaction may abort it if they so decide, and the implementation (either of the shared object itself, or of some general transaction mechanism, making also possible transactions involving more than one shared object) will make sure that all the operations comprising the transaction are undone in case of an abort, or committed together in case of successful completion.

There are important differences between the contexts for which the use of atomic transactions and of conversations are usually envisaged. First, transactional servers are typically seen as guardians for well-identified, permanent, externally visible (in some limited, controlled form) data structures. The server is programmed to explicitly perform updates on these data, and does so using temporary execution threads and variables. The programmer of the server component is aware of the effect of requested operations on the encapsulated data, and can therefore explicitly program concurrency control, and forward and backward recovery. With conversations, backward recovery of the whole state of processes is assumed, implying
no explicit distinction between permanent important data and temporary variables, and no well-defined connection between messages received and consequent changes in the state of the component. Second, transactions normally have an initiator component, which is a natural coordinator for their execution and commit. All the activities carried out by different components within a transaction (as servers and/or as clients) are connected in a call-reply tree structure, which provides a natural error reporting hierarchy. With conversations, the participant processes are peers, and the decision to commit or abort is a matter of distributed consensus.

How can conversations among processes and transactions on data objects (or server components) be combined in the same system? [Mancini 1989, Shrivastava 1987] show that there is a "duality" between the two "philosophies", and that policies designed for either can be translated into the other. We show a way of combining components written according to both philosophies in the same system. This would allow the use of conversation-based coordination where appropriate, but also allow "conversational" components to dynamically decide interactions with server components. Also, server components allow greater efficiency by implementing specific concurrency control policies, which can be much finer and allow greater concurrency than the generic, strict control imposed by conversations: a component can only be in one conversation (at each level of nesting) at a time, while transactional components may, if allowed by their concurrency control policy, be engaged in many transactions.

Assume a system made up of components (processes) that synchronize their checkpointing/recovery via conversation-like mechanisms, and others (server components) that implement an atomic transaction mechanism: they accept requests to start, abort and commit transactions and serve them according to some policy that ensures proper concurrency control among competing transactions. We assume that the run-time support accepts requests such as "enter conversation" and "commit conversation" and that the division of components into conversational and transactional is known to the run-time support. Besides, call-reply exchanges between clients and servers are implemented by messages, and the support is able to recognize and capture transaction control messages (open, abort, commit).

If a component requests operations on a transactional component while engaged in a conversation, it must be ensured that these operations can be undone if the conversation were to be rolled back. So, a first rule for the programmer of the client component is:

*Any operation on a transactional component that is performed within a conversation must be part of a transaction started within the innermost current conversation (we say that that conversation creates the transaction).*

The support must block, and signal as errors, all calls that are not so bracketed. It must also delay the commit of a transaction, lest a subsequent roll-back of a conversation leave the effects of the transaction standing. This delay not only allows roll-back, but prevents the transactional object from smuggling information between conversations. How long must the commit of the transaction be delayed? If the conversation that created the transaction is the outermost conversation, then the commit of the transaction must be delayed until the conversation itself commits: at that point, all actions of that conversation may be made visible to components outside it. However, more complex situations may exist. In Fig. 5, transaction T2 is created by conversation C3. If T2 were committed at the end of C3, its results would be
visible to component P, outside C2. If then C2 rolled-back, this visibility would become a case of information smuggling through the borders of C2. So, T2 must only be committed when C2 commits. Likewise, if T1 did not exist (that is, if T2 were the outermost transaction on component Q), T2 should not be committed until C1 commits. We can say that the commit of a transaction T must be delayed until the commit of its associated conversation, which can be found according to the following algorithm. The conversation C associated with T is the innermost conversation such that: i) C encloses the opening of T; and ii) either C is a top-level conversation, or the conversation immediately enclosing C has an associated transaction. Notice that at run-time the associations between conversations and transactions are easily determined as transactions are opened: an appropriate naming scheme for conversations can provide the nesting information needed.

To prevent information smuggling between conversations, a second rule must be respected by the client programmer:

*If a transaction has multiple participants, these must belong to the same enclosing (innermost) conversation.*

This can easily be enforced by the run-time support; actually, if for a second process to join a transaction initiated by another process there must be a communication between the two processes, the very protection mechanism of conversations ensures that the rule is respected.

Last, the fact that the commit of a transaction is delayed until the commit of its associated conversation implies a third rule for the programmer:

*Two transactions at the same level of nesting can be opened within the same conversation only if they are known not to interfere with each other, otherwise deadlock may ensue.*

(This rule does not address the possibility of deadlock on shared objects in general, but just that which may arise from the forced delay in committing transactions due to the mechanism for coordination with conversation-based components). Fig 5db shows some applications of these rules.
A variation on this scheme would guarantee, by some complication in the support, that roll-back from conversations is possible even if the programmer of the client did not see the need for enclosing the operations in a specific transaction. In this variation, if components engaged in a conversation request the services of a server component, the run-time support must label the request as coming from a conversational component, and include a (run-time
For each message sent by a conversational component, the run-time support must check whether the destination is a conversational component in the same conversation, or a transactional component (these are the only two legal cases). Since a check that the destination of a message is a participant in the same conversation as the sender is needed anyway, this is not a particularly expensive complication.

The first request with a new conversation identifier is then interpreted as a request to start a new transaction, and all the following requests with the same identifier as belonging to the same transaction. If the first request to the transactional component is from inside an inner nested conversation, then implicit transactions must be created for each enclosing conversation (an appropriate naming scheme for conversations could provide the nesting information needed). The run-time support must also keep track of all the transactional components called during a conversation, and send an abort or a commit depending on whether the acceptance test fails or succeeds. All transactions that are explicitly requested must be treated as sub-transactions of the last implicit transaction created by the conversation coordination mechanism. Both implicit and explicit transactions may be nested, but the programmer must be aware that the nesting must be a total ordering among all transactions of both types in which the component is engaged at any moment. Of course, a programmer calling a transactional component would typically have to bracket operations on a transactional component between explicit transaction requests on the component (to declare which data items must be locked, for instance). So, the normal cases would be that either the first request from within a conversation is an "open transaction", or, if it is not, it creates a subtransaction inside an already open transaction. Fig. 6al illustrates the application of this rule. The programmer must be aware that opening a nested conversation implicitly creates transactions which may cause interference and deadlock (a typical conservative concurrency control policy for a transactional object would lock all the data involved in the next enclosing transaction).
4 Combining components with different recovery provisions

The example in the previous section shows some of the problems in combining components that follow different schemes for recovery and/or different design styles. If the two component designs use different notations, or sets of primitive concepts (as remote procedure calls vs simple unidirectional messages) it is necessary to define a correspondence between the two sets of primitive concepts, typically by interpreting both in terms of an enlarged set of lower level concepts. In the example, each remote procedure call was trivially mapped into two messages. This mapping must be extended to all the extensions made to the notations used in order to support fault-tolerance policies: in our example, it was necessary to describe the reactions of the run-time support to transaction control messages. Last but not least, the additional restrictions imposed on the designers of the heterogeneous communicating...
components must be spelled out explicitly. In this section, we explore the possibility of providing standard solutions for these problems.

To visualize the problem, let us consider the following example: a subsystem B, capable of backward error recovery, sends messages to a subsystem F, that is only capable of forward recovery (one could always choose to build F on an interpreter supporting checkpointing and roll-back: we assume that for F this is too expensive, or the timing requirements for F do not allow roll-back, etc.).

Then, two solutions are possible. In the first, B can be made fail-silent and deterministic, and its roll-back, with the replay of previously sent messages, can be masked to F by the support (and even this can be avoided if all operations on F are idempotent). Otherwise, application-specific measures can be taken to cope with the two problems of correctness of F (in spite of errors propagated by B) and consistency between B and F. A first option is simply to raise a special exception in both B and F (B and F would be part of a same "atomic action" in a scheme such as proposed in [Campbell 1986]), which have reasonable means for independently coping with the situation. Other options are:

- B is rolled back and retried; the interpreter "pretends" that B is fail-silent and deterministic, replays to it the previous input messages and hides its duplicate output messages to F. This could work, typically, if F is capable of tolerating occasional erroneous input messages (e. g., B gives F the instant value of a physical variable on which F does some time-related filtering. Any individual erroneous message causes only a transient error in the state of F) and if B received messages that were not functions of its own outputs F did not reply to B's erroneous outputs.

- B is rolled back and its repeated send and receive operations are visible to F. This may work if: i) F may ignore messages that arrive at an improper time, and is reasonably immune, as in the previous case, from errors propagated by B; or ii) the repeated messages can correct the effect of the previous erroneous messages, and possible replies by F will therefore provide correct outputs to B. For instance, F might be a database (without explicit undo or roll-back operations). The message from B to F was a query. If it was wrong, the answer was also presumably different from what B needed. Repeating the query will make sure that B proceeds correctly while F is unaffected.

This informal discussion shows that many combinations are possible between the capabilities for error confinement and recovery implemented in communicating components. To combine these components together, and obtain a predictable, sound behaviour in case of errors, one needs either to know the details of the behaviour of the interacting components, or at least to be able to classify them into categories for which standard solutions are known. Some properties that are relevant for such a classification are:

- the types of recovery behaviour of which the component is capable: interpreter-supported roll-back of the whole state (with the fail-silent/deterministic property or not), application-implemented roll-back, forward recovery.

- behaviour with respect to inputs: a component may be able to:
• recognize and possibly signal input messages with erroneous contents;
• tolerate occasional wrong inputs (which do/do not cause it to produce wrong outputs in the short term);
• tolerate occasional missing inputs, or input messages that are duplicates of messages previously received, without "falling out of synch" with the rest of the system.

For instance, a component that can recognize wrong inputs and can also tolerate occasional missing or duplicated inputs, but cannot do backward recovery on request, can be programmed simply to ignore the wrong inputs. Then, it could be combined with any component that performs backward recovery by simply instructing the interpreter that messages issued by the latter component during a retry must be propagated as normal messages.

Producing a classification and a corresponding set of rules that is both general enough and practically useful is not trivial. The designer and the run-time support must not be required to deal with too many classes of components (which would create complexity hard to manage both by hand and in programming a run-time support). At the same time, the classes that are chosen must be representative of the meaningful ways components can be built, though necessarily prohibiting some of them. Rules must be given to determine which components can be neighbours in the communication graph of a system (given the capabilities of the interpreter), and which additional constraints the neighbourhood imposes on the programmers of each individual component; last but not least, it would be very useful to be able to infer the class of a component from the properties of its communicating subcomponents, so that the same rules apply to views of the system at different levels of detail.

Conclusions

We have argued the importance of application-level fault tolerance, and of a proper discipline in its application. This discipline, while imposing enough restrictions to keep the design manageable, should allow more flexibility in combining basic schemes than usually considered in the literature. We have provided an example, in the form of programming rules, and extensions to a run-time support, that would allow components using coordinated backward recovery via conversations to use the service of server components providing atomic transactions. Last, we have outlined a scheme for classifying components according to their fault tolerance characteristics, in order to establish rules for their correct interfacing.

While our example is a viable technical proposal, the issue of classification of components and selection of design rules obviously requires much further study: there are implications for, and constraints from, both the facilities that an underlying virtual machine should provide to allow the interfacing of different classes of components, and the complexity of the notation used in the design under consideration.

Our examples consider one type of component, with simple message passing or call-reply pairs as their communication mechanisms. Many operating systems allow more variety of basic concepts for the application designer to use, and it is not clear to which degree designers
can be asked to give up that variety for the sake of simpler structuring. What degree of
generality allowed by a notation, and possibly supported by extensions to a programming
language, can be usefully exploited in design is always debatable. The practicality of a design
notation can be evaluated only by experiment, trying to apply it in the design of actual
applications. To this aim, we are planning exercises in the high-level design of practical
applications with application-level fault tolerance.

One aspect of many real-world systems that must be considered for generality is that of
dynamicity: the possibility for components to be created and destroyed during operation.
Dynamics creates specific problems, starting with the degree to which it should be allowed.
Completely static systems are easiest to design reliably, but have obvious limits in power and
efficiency. Steps towards completely dynamic systems are different modes of operation (for
instance, take-off, cruise and landing for an autopilot), components that can be replicated in
numbers variable at run-time (for instance, processes created to handle outstanding
transactions), dynamic creation hidden inside components (as in processes with fork and join
operations), and upgrade of components during operation. If such limited degrees of
dynamicity were deemed sufficient, it would be much easier to devise notations and design
rules that would help to keep designs manageable.

Another issue for further study is how consideration of the hardware configuration of
the system can be orderly included in designing application-level fault tolerance. While the
latter is certainly applicable against hardware faults, its effective use seems to require an
integration of hardware and software considerations in the design phase.

A last important issue is evaluation: simple fault tolerant structures typically allow the
simple application of structural dependability models to the whole system. The proposals in
this report aim at allowing effective design of more complex systems, in cases where such
complexity is required by the applications. These structuring rules should help in keeping
complexity under control, so that a designer can still forecast the behaviour of the system, but
they cannot be expected to completely avoid its negative effect on dependability. Insofar as
design complexity is introduced by the use of heterogeneous fault tolerance provisions, there is
a trade-off, which has to be studied, between the pursuit of actual dependability of a system,
through flexible application of the most effective techniques for each component, and the ability
to evaluate it.

Acknowledgements

The authors would like to acknowledge the critical contributions of their colleagues in the
ESPRIT PDCS project.

Part of this work will appear as an article in the Proceedings of the IEEE 10-th
Symposium on Reliable Distributed Systems, Pisa, Italy, 30 September-2 October 1991.

References


