Fault Tolerant Concurrent C: A Tool for Writing Fault Tolerant Distributed Programs

R. F. Cmelik N. H. Gehani W. D. Roome

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Concurrent C is a superset of C that provides parallel programming facilities. Our local area network (LAN) multiprocessor implementation has led us to explore the design and implementation of a fault tolerant version of Concurrent C, FT Concurrent C. FT Concurrent C allows the programmer to replicate critical processes. A program continues to operate with full functionality as long as at least one of the copies of a replicated process is operational and accessible. As far as the user is concerned, interacting with a replicated process is the same as interacting with an ordinary process. FT Concurrent C also provides facilities for notification upon process termination, detecting processor failure during process interaction and automatically terminating orphan processes.

In this paper, we discuss the different approaches to fault tolerance, describe the considerations in the design of FT Concurrent C, and, finally, present a programming example.

1. Introduction

Concurrent C is a superset of C that provides parallel programming facilities [10]. Concurrent C has been implemented on both uniprocessors and multiprocessors [6]. The local area network (LAN) multiprocessor implementation has led us to explore the design of a fault tolerant version of Concurrent C, *FT Concurrent C*, which can be used to write portable programs that will continue to operate with full functionality despite the failure of some processors.

Fault tolerance is particularly important for computer systems engaged in "continuous" real-time applications such as switching, process control, on-line databases and avionics. Unfortunately, even the most reliable components are susceptible to failure. The need for fault tolerance arises if the reliability of the system is to be greater than that of its components [9]. SIFT [22] has successfully demonstrated the building of a reliable avionics system by replicating software. Replication has also been used by others to provide fault tolerance; for example, replication of complete Ada programs with tasking [15], replicated procedure calls [7], CSP processes with voting [14] and CSP processes without voting [11]. Compared to SIFT in which communication with the replicated processes is limited to broadcasts at the end of their activities [15], these systems provide more extensive interaction with replicated processes.

As done in the above mentioned systems, our goal is to extend the notion of the SIFT fault tolerance to a distributed programming language; in our case this language is Concurrent C, which runs on different types of multiprocessors. Cost is an important factor in the design of fault tolerant systems. To reduce the cost of fault tolerance, our emphasis is also on the design of fault tolerant programs with selective fault tolerance, i.e., programs in which only the critical parts are made fault tolerant. FT Concurrent C supports selective fault tolerance by allowing the programmer to replicate critical processes. FT Concurrent C also provides facilities for notifying interested processes of process failure, detecting processor failure during process interaction and automatic termination of orphan processes.

In this paper, we will discuss fault tolerance, issues in the design of FT Concurrent C, and, finally, we will show you a programming example with selective fault tolerance.

2. Fault Tolerance

The term "fault tolerance" has been used in a variety of ways and contexts. We are interested in software fault tolerance, that is, the use of software to tolerate failure of the underlying hardware and to tolerate software design faults [16]. Our initial research goal is to provide high-level architecture-independent facilities for allowing a program running on multiple processors to continue operating despite the failure of one or more of the processors.

2.1 Fault Tolerance Approaches

Fault tolerance can be provided at a variety of levels in a system. One is the *hardware approach*, which uses redundant, fault tolerant hardware. The advantage is that this usually does not require additional programming effort at the user level, e.g., FTMP [19]. However, this approach usually requires specialized hardware.

An alternative is the *transparent approach*, in which the underlying operating system transparently provides fault tolerance to application programs. This involves duplicating processes (as in the Tandem system [4], the Auragen system [5], and the fault tolerant Ada architecture [15]) and/or saving state on stable memory. This approach works on a variety of hardware, and the application programmer gets fault tolerance with no additional programming effort. Typically, the state of each process is checkpointed periodically. To minimize delays due to checkpointing on stable store, Strom & Yemini [20] and Johnson & Zwaenepoel [12] have proposed new techniques based on asynchronous checkpointing.

A third alternative is the *fault tolerant tools approach*, which provides programming facilities that allow an application programmer to write programs with critical parts made fault-tolerant. Essentially this approach is a tradeoff: in exchange for some additional programming effort, the programmer gets efficient fault tolerance. The tools approach can be based on stable storage and checkpointing [13, 17] or on replication of program components such as objects [3] or processes [11, 14].

2.2 Our Approach to Fault Tolerance

We have chosen the tools approach for FT Concurrent C because we feel that it allows selective system fault tolerance to be achieved at a reasonable cost. And we feel that the system

designer is the person who can best specify the parts of the system that should be fault tolerant. FT Concurrent C uses the "replicated process" model. A *replicated process* consists of a set of "identical" processes called *replicas*. Critical processes are replicated, and the replicas are normally placed on different processors. The program will continue to operate as long as at least one of these replicas is alive. Note that all replicas execute the same algorithm and they run on identical processors.

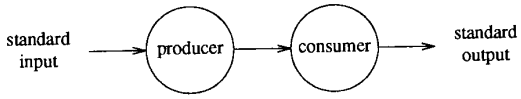
We shall use the unadorned term *process* to refer to an ordinary (non-replicated) process. At times, when the distinction is not necessary, we will be informal and we will use the term "process" to refer to all of the above kinds of processes.

3. Brief Summary of Concurrent C

A Concurrent C program consists of a set of components, called *processes*, that execute in parallel. Two processes interact by first synchronizing, then exchanging information and, finally, by continuing their individual activities. This synchronization or meeting to exchange information is called a *rendezvous*. Transfer of information during a rendezvous is unidirectional—from the message sender to the receiver. Concurrent C uses the *extended rendezvous* or *transaction* concept to allow bidirectional information transfer during the rendezvous (a Concurrent C "transaction" is a process interaction, and is similar to a remote procedure call). After a transaction has been established, the process requesting service is automatically forced to wait until the server completes the requested transaction; the transaction results are then sent back to the waiting client.

We introduce the basic concepts of Concurrent C by presenting a simple program which reads data, processes it, and then prints the results. The data is a stream of characters, and the "processing" consists of converting all lower-case characters to upper case.

The Concurrent C program has two processes. The "producer" process reads data from the terminal and sends it to the "consumer" process, which converts each character to upper case, and then prints it on the standard output:



Here is the Concurrent C program:

```
process spec consumer() { trans void send(int); };
process spec producer(process consumer);
process body producer(cons)
{   int c;
    while ((c = getchar()) != EOF) cons.send(c);
    cons.send(EOF);
}
process body consumer()
{   int ch;
    while (1) {
        accept send(c) ch = c;
        if (ch == EOF) break;
        if (isupper(ch)) ch = toupper(ch);
        putchar(ch);
    }
}
main()
{   process consumer q;
    q = create consumer(); create producer(q);
}
```

There are specifications for two process types: `consumer` and `producer`. The `consumer` process has a transaction named `send` which can be called by other processes to send a character to a process of type `consumer`. The body of `consumer` contains an `accept` statement which is how a process accepts a transaction call. Finally, the `main` process creates instances of `producer` and `consumer` processes, and give the "process id" of the `consumer` process as an argument to the `producer` process.

The `producer` process reads characters from the standard input and calls transaction `send` to send these characters, one at a time, to the `consumer` process by calling transaction `send`, e.g.,

```
cons.send(c);
```

where *c* is the character being sent.

The `consumer` process accepts each transaction call, saves the passed character in a local variable, converts that character to upper case if needed, and then prints it.

4. FT Concurrent C

FT Concurrent C is a superset of Concurrent C [10]. The FT Concurrent C extensions include facilities for creating replicated processes, detection of failure of server processes during inter-process communication, informing interested processes about the death of specific processes, and automatic termination of slave processes when they become orphans.

4.1 Assumptions

FT Concurrent C makes the following assumptions:

1. Processor failure can be detected.
2. The processors are homogeneous and fail by stopping.
3. Each message is either transmitted correctly or not at all.
4. All working processors can communicate with each other.

Thus we assume that a failed processor does not produce incorrect responses [18] (we do not consider malicious "Byzantine" failures), and we assume that failures do *not* cause the processors to "partition" into separate groups, each of which thinks that it is the sole survivor.

4.2 Replication Model

In defining the semantics of interaction with a replicated process we have two choices: to use a voting scheme [14] or to just take the response of the first replica and discard the responses of the other replicas. The "first-response" approach just protects against processor failures; this is similar to the fault tolerance provided by most "check-pointing and rollback" systems [13, 21]. The voting approach can also survive some malicious processor failures, although $2n+1$ replicas would be required to survive n such errors.

Although the first-response approach does not provide as much fault protection as the voting approach, it requires less redundancy (only one active replica is needed for full functionality), and the program as a whole may run faster, because a process interacting with a replicated process does not have to wait for all the replicas to respond. That is, the replicas do not run in lockstep. Although replicas can fall behind in execution, they must agree upon the order in which external events take place. For example, a replica can accept a message X even though the other replicas may be lagging behind. However, before accepting X, a distributed consensus protocol is used to ensure that each replica will, when it is ready to do so, accept X. Eventually, of course, if a replica goes too far ahead, past implementation limits, then it will be suspended while the others catch up.

We have selected the first-response approach for FT Concurrent C. As in the Circus system [7], we may eventually allow the programmer the option of using either the voting or first-response schemes. Note that an FT Concurrent C program could be ported without any change from a FT Concurrent C with a first-response implementation to one that uses voting.

4.3 Replicated Process Behavior and Programmer Responsibility

A FT Concurrent C replicated process behaves like a single process. Its replicas cannot, in general, be referenced individually. Interaction with a replicated process automatically implies interaction with all of its replica processes. A copy of a request to a replicated process is sent to all replicas, and all replicas act on each request. All replicas generate replies; the first reply is returned to the requester, and the others are discarded (remember that we assume that a processor fails by stopping, so the replies from all working replicas should be identical). FT Concurrent C does all of this automatically.

The programmer must do two things. First, the programmer must analyze the program and decide which processes must be replicated. And second, the programmer must ensure that all replicas of a replicated process have the same external behavior. That is, if several replicas of this process are created, and if identical transaction calls (i.e., input messages) are presented to each replica in the same order, then each replica must generate the same transaction calls (i.e., output messages), in the same order.

In practice, making the replicas exhibit the identical behavior is rarely a problem. There are two sources of possible "variant" behavior. One is explicit in the process, as written by the programmer. For example, a process can use a random number generator to decide to make replicas behave differently.

The other source of variant behavior is inherent in the definition of Concurrent C which allows processes to execute non-deterministically. For example, a process can wait for the first of several events (transaction calls, delays, etc.); if several are available, the Concurrent C implementation can pick any of them. In FT Concurrent C, we avoid this form of variant behavior by picking the same event in each replica. This is done automatically. That is, whenever the FT Concurrent C implementation can make a non-deterministic choice, it guarantees that it will make the same choice for each replica.

4.4 No-Fault Call Operator

Transaction calls are used in Concurrent C for process interaction. The transaction call syntax is extended by adding the "no-fault" transaction call operator, `??`:

transaction-call `??` *fault-expr*

If the called process accepts this call and returns a value, then that value becomes the value of the `??` operator, and *fault-expr* is not evaluated. However, if the called process has terminated, then *fault-expr* is evaluated and its value becomes that of the expression. In an ordinary transaction call, if the called process fails or does not exist, the FT Concurrent C run-time system prints an error message and kills the calling process. Note that this operator is primarily useful for calling non-replicated processes.

4.5 Death Notices

The "death notice" mechanism allows a process to request that the FT Concurrent C run-time system generate a transaction call when (and if) a process terminates. For example, suppose a client process calls one of a server's transactions to allocate a resource, and when done, the client calls another transaction to release the resource. If the client process terminates in between, the server would not be able to reclaim the resource. The server can avoid this by asking to be notified if the client process terminates.

To request a death notice for process p, a process calls the built-in function `c_request_death_notice` to request that a specific transaction be called when p terminates. If p is already dead or does not exist, then the transaction call is generated immediately. If p is a replicated process, the death notice is sent only if all of the replicas die. A process can withdraw a death notice request by calling the built-in function `c_cancel_death_notice`.

4.6 Creating Replicated Processes

Creating a replicated process with n copies, each with identical arguments, is similar to creating an ordinary process:

```
create [slave] process-type(arguments)
       [copies(n) | processor(n1, n2, ...)]
```

The `create` operation is successful if at least one replica can be created. If successful, the `create` operator returns a single process identifier which identifies the entire *set* of replicas; otherwise, it returns a null process id. In general, it is not possible to refer to an individual replica. If the processors are not explicitly specified, then the replicas are automatically placed on different processors (if possible).

The newly created process can be designated as a "slave" of the parent process. If the parent process terminates abnormally, then FT Concurrent C guarantees that all of its "slave" processes will be killed. If, on the other hand, the parent process terminates normally, then its slaves are "freed" and can run independently. To see why this is useful, suppose that a process wants to create a number of replicated "worker" processes. If the parent process terminates abnormally, then the worker processes will become orphans and may stay around doing nothing. However, by designating the created processes as slaves, the parent can ensure that if it terminates abnormally, then the slave processes will be terminated automatically.

When a replicated process creates another process—replicated or not—only one instance of the process is really created. The first replica to execute a create operation actually creates the new process. When the other replicas execute the corresponding `create` operator, no new process is created; instead, FT Concurrent C returns the process identifier that was given to the first replica.

In most cases, all replicas of a replicated process get the same process arguments. However, sometimes it is useful to give them different arguments. For example, if we build a replicated file system manager, with a separate disk for each replica, then we need to tell each replica which disk to use. An alternative form of the `create` operator allows each replica to be given different arguments:

```
create [slave]
       (process-type(arguments) [processor(n1)],
        process-type(arguments) [processor(n2)],
        ... )
```

It is the programmer's responsibility to ensure that the different replicas still behave in the same manner. In particular, interactions of each replica with other processes must occur in the same sequence and must be identical.

4.7 Interacting With Replicated Processes

4.7.1 Calling a Replicated Process. When a client calls a replicated process, FT Concurrent C automatically sends the call to all the replicas. When the first replica completes the call, FT Concurrent C activates the client process and gives it the value returned by that replica. The other replicas will eventually accept this transaction and generate replies, but FT Concurrent C will automatically discard those extra returns.

4.7.2 Within A Replicated Process. Each replica must execute similar, if not identical code. In particular, each replica must perform the same sequence of identical process interactions. For example, each replica *must* issue the same sequence of transaction calls to the same processes. Such interactions include transaction calls, accepting transaction calls, returning transaction results, waiting for a set of events, and creating processes.

Some Concurrent C statements are non-deterministic. For example, the `select` statement non-deterministically waits for one of a set of events. FT Concurrent C ensures, by using a distributed consensus protocol [1], that it makes the same choice for each replica (even if some of the replicas fail).

4.7.3 Transaction Calls From A Replicated Process. When a replicated process calls another process, each replica must make that transaction call. Only the first call issued by one of the replicas is really given to the called process; the other calls are discarded. The distributed consensus protocol [1] ensures that each replica gets the same result; the returned value is saved, and is given to the "lagging" replicas when they issue that transaction call.

FT Concurrent C also ensures that each replica will take the same alternative in a timed transaction call (transaction call with a time out) or a no-fault (`??`) call.

4.7.4 Accept And Treturn Statements. An `accept` statement accepts transaction calls. By default, these messages are accepted in FIFO order, but the `suchthat` and `by` clauses can be used to accept messages in another order. The `treturn` statement returns a result for the transaction. When a replicated process executes an `accept` statement, FT Concurrent C ensures that each replica accepts calls in the same order.

4.7.5 Select Statements. The `select` statement non-deterministically waits for one of a set of events. Examples of events are accepting a pending transaction call, the arrival of a new call, and timeouts. These events are specified as alternatives, and each alternative can be preceded by a boolean guard expression. Only those alternatives that are not preceded by a guard expression or those with a guard expression that evaluates to true are considered in the execution of a `select` statement.

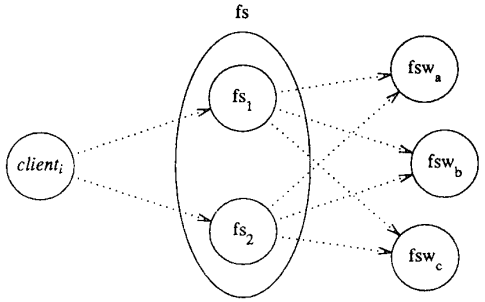
When a replicated process executes a `select` statement, FT Concurrent C ensures that all the replicas execute the same alternative.

5. Example: A Robust File Server

This section shows how the facilities provided by FT Concurrent C can be used to write a robust file server. This file server maintains several copies of each file, on different processors, and performs every file operation on all copies of the file. As long as the file server is running and as long as the processors (and disks) on which the files are actually placed are operating, file operations will not be lost.

The file server will be implemented as a replicated FT Concurrent C process. The file server process will create one worker process for each copy of each file; each worker process will do all operations on its file copy. That is, the file server manually replicates the worker processes. We do that so that each worker process may return a different result based on the actual file operation. That allows the file server to decide what to do if one of the worker processes discovers an error when accessing its file copy.

Here is a process diagram of the file server. The ellipse represents the replicated file server process, and the circles within it are the replicas. The "fsw" processes are the worker processes.



Here are the specifications of these processes:

```
typedef struct {
    int nbuf, valid;
    char buf[NBUF];
} Buf;
process spec fsWorker() {
    trans int  open(Buf,int,int);
    trans int  close();
    trans int  write(Buf);
    trans Buf  read(int);
};
process spec fs() {
    trans int  use(int);
    trans int  open(Buf,int,int);
    trans int  write(int,Buf);
    trans Buf  read(int,int);
    trans void death(process anytype);
};
```

The `main` process creates the replicated file server. The file server process loops accepting `use`, `open`, `close`, `read` and `write` requests for manipulating files. The `use` requests (from `main`) inform the file server on which processors to perform the file operations.

Upon receipt of an `open` request, the file server looks for an unused entry in its file table. Here it records the file name and the pid of the calling process (client). A death notice is requested for the client. The pids of the file server workers (of type `fsWorker`) it creates are also saved. Each worker is then called with the file name and other `open` parameters. If the `open`s are successful, a non-negative integer value is returned and must be used in subsequent `read`, `write`, or `close` calls to the file server.

The `close` request causes the file server to call the file server workers' `close` transactions, after which the workers terminate,

and to free the file table slot, and to cancel the death notice for the client.

The `read` and `write` requests cause similar operations to be carried out on each worker associated with the file. If the file server receives a death notice for one of its clients, it proceeds as for a close of the associated file. Calls to the file server workers use the `??` operator to ensure that failure of the workers does not cause termination of a file server process copy. If a worker terminates, work proceeds with the remaining workers. If all workers for a file are lost, subsequent file server requests for that file cause an error indication to be returned.

File server workers are created for each file server on the specified processor. The `fsWorker` first waits for an `open` transaction. It then loops accepting `read` and `write` requests, which it carries out on the previously opened file. Upon accepting a `close` request, it closes the file and terminates.

Here is the body of the `fsWorker` process:

```
process body fsWorker()
{   int fd = -1;
    accept open(path, flags, mode) {
        fd = open(path.buf, flags, mode);
        treturn fd;
    }
    if (fd < 0)
        return;
    while (1)
        select {
            accept close()
                treturn close(fd);
            break;
        or  accept read(n) {
                Buf B;
                B.nbuf = read(fd, B.buf, n);
                B.valid = 1;
                treturn B;
            }
        or  accept write(B)
                treturn write(fd,B.buf,B.nbuf);
        }
}
```

And here is the body of the `fs` process:

```
process body fs()
{   process anytype caller; trans void (*tp)();
    other declarations
    tp = ((process fork)c_mypid()).death;
    while (1)
        select {
            accept use(procnum)
                save processor number
        or  accept open(path, flags, mode) {
                caller = c_caller_pid();
                c_request_death_notice(caller, tp);
                find empty slot in file table, save caller pid;
                create fsWorkers, save pids, call open trans;
                if any successful, return file table index;
                if not, return -1;
            }
        or  accept close(fd)
                {call close for all workers; free table slot; ... }
        or  accept read(fd, size)
                {call read for all workers; ... }
        or  accept write(fd, B)
                {call write for all workers; ... }
        or  accept death(pid)
                {find client in table; do "close"; }
        }
}
```

To illustrate the use of the no-fault transaction call operator, consider the following code that sketches the body of the transaction `read` in the file server process `fs`:

```
accept read(fd, size) {
    Buf good, bad, tmp;
    bad.valid = 0; good.valid = 0;
    for all workers fsw associated with the file fd {
        tmp = fsw.read(size) ?? bad;
        if (!tmp.valid)
            mark fsw as dead;
        else if (!good.valid)
            good = tmp;
    }
    treturn good;
}
```

6. Conclusions

FT Concurrent C is a tool for writing fault tolerant distributed programs. Critical program components (processes) can be made fault tolerant by replicating them. All interaction with the replicated processes is managed by the FT Concurrent C run-time system. An important advantage of the replicated process approach is that it does not add too much complexity to the already difficult task of writing distributed programs. In fact, FT Concurrent C programs look very much like their non-fault tolerant counterparts.

Fault tolerance does not come for free; Dahbura, Sabnani & Hery [8] advocate the use of spare capacity of a system for fault tolerance. A price has to be paid for the replicated processes because of the

1. extra transaction calls,
2. synchronization between copies of the replicated process (the "distributed consensus" protocol),
3. changes to the data structures required to support replicated processes.

The first two items are the major contributors to the extra execution overhead due to process replication. The overhead is directly related to the amount of interaction with replicated processes and the number of replicated processes. Consequently, if these items are kept to a minimum, then the overhead due to process replication will be small. Note that experiments by Cooper [7] show that the overhead per interaction increases linearly with the amount of replication.

The underlying inter-processor communication medium can help reduce the cost of fault tolerance. For example, if the hardware supports broadcasting, then this will reduce the consensus cost. Moreover, if the broadcast messages are received in the same order by all the processors as in a LAN, then this will reduce the need for using the distributed consensus protocol. The distributed consensus overhead can be trimmed substantially by using a special purpose communication chip [2].

An important issue is the resurrection of the failed copies of a replicated process. In some cases, it may be sufficient to copy the failed process to be resurrected by simply copying an operating copy along with its data. However, if these processes have modified the environment, e.g., disks, then resurrecting the failed process may require additional "environmental" support.

One limitation of the replicated process approach is that complete failure of the system, such as that caused by power outage will require restarting from scratch.

We are building a prototype implementation and with it we hope to better evaluate the replicated process model and to gain some real experience in writing fault tolerant programs.

Acknowledgements

Discussions with A. Asthana were very beneficial. We also appreciate the suggestions and comments of T. A. Cargill, A. T. Dahbura, B. W. Kernighan, M. Merrit, D. E. Perry, and M. E. Quinn.

References

[1] Arevelo, S. and N. Gehani 1987. Replica Consensus in Fault Tolerant Concurrent C. AT&T Bell Laboratories.

[2] Asthana, A. 1987. A Fault-Tolerant Architecture with Dynamically Reconfigurable Redundant Partitions. AT&T Bell Laboratories.

[3] Birman, K. P., T. A. Joseph, T. Raeuchle and A. E. Abbadi 1985. Implementing Fault-Tolerant Distributed Objects. *IEEE TSE*, vSE-11, no. 6 (June), pp. 502-508.

[4] Bartlett, J. F. 1981. A NonStop Kernel. *Proceedings of the 8th ACM Symp. on Operating System Principles, Operating Systems Review*, v15, no. 5.

[5] Borg, A. Baumbach, J. and S. Glazer 1983. A Message System Supporting Fault Tolerance. *Proceedings of the 9th ACM Symp. on Operating System Principles, Operating Systems Review*, v17, no. 5.

[6] Cmelik, R. F., N. H. Gehani and W. D. Roome 1987. Experience with Multiple Processor Versions of Concurrent C. To be published in *IEEE TSE*.

[7] Cooper, E. C. 1985. Replicated Distributed Programs. *Proceedings of the 10th ACM Symp. on Operating System Principles, Operating Systems Review*, v19, no. 5.

[8] Dahbura, A. T., K. K. Sabnani and W. J. Hery 1986. Spare Capacity as a Means of Fault Detection and Diagnosis in Multiprocessor Systems.

[9] Drummond 1986. Impact of Communication Networks on Fault Tolerant Distributed Systems. PhD Thesis, TR86-748, Cornell University, Ithaca, NY 14853.

[10] Gehani, N. H. and W. D. Roome 1986. Concurrent C. *Software—Practice & Experience*, v16, no.9, pp. 821-844.

[11] Jalote, P. and S. K. Tripathi 1986. Fault Tolerant Computation in Synchronous Message Passing Systems. University of Maryland, College Park, MD 20742.

[12] Johnson, D. B. and W. Zwaenepoel 1987. Sender-Based Message Logging. *FTCS 17 Digest of Papers*, Pittsburgh, PA.

[13] Liskov, B. and R. Scheifler 1982. Guardians and Actions: Linguistic Support for Robust, Distributed Programming. *ACM TOPLAS*, v5, no. 3 (July 1983), pp. 381-404.

[14] Mancini, L. A Watchdog Processor Based General Rollback Technique with Multiple Retries. *IEEE TSE*, vSE-12, no. 1 (January).

[15] Melliar-Smith, P. M. and R. L. Schwarz 1984. A Fault-Tolerant Ada Architecture. *Proceedings of the 4th Jerusalem Conference on Information Technology*, Jerusalem, Israel.

[16] Randell, B. 1975. Software Structure for System Fault Tolerance. *IEEE TSE*, vSE-1, no. 2 (June), pp. 220-232.

[17] Reed, D. P. 1983. Implementing Atomic Actions on Decentralized Data. *ACM TOCS*, v1, no. 1, pp. 3-23.

[18] Schlichting, R. D. and F B. Schnedier 1983. Fail-Stop Processors: An Approach to Designing Fault Tolerant Computing Systems. *ACM TOPLAS*, v1, no. 3 (August), pp. 228-238.

[19] Smith III, T. B. and J. M. Lala 1984. Development and Evaluation of a Fault-Tolerant Multiprocessor (FTMP) Computer. Tech. Report, Charles Stark Draper Labs.

[20] Strom, R. E. and S. Yemini 1985. Optimistic Recovery in Distributed Systems. *ACM TOCS*, v3, no. 3 (August), pp. 204-226.

[21] Svobodova, L. 1984. Resilient Distributed Computing. *IEEE TSE*, vSE-10.

[22] Wensely, J. et al. 1978. SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control. *Proceedings of the IEEE*, v60, no. 10, pp. 1240-1254.