

Fault Injection Campaign for a Fault Tolerant Duplex Framework

Gian Franco Sacco¹, Robert D. Ferraro¹, Paul von Allmen¹,
Dave A. Rennels^{1,2}

¹*Jet Propulsion Laboratory, California Institute of Technology,
Pasadena, California 91109*

²*Computer Science Department, UCLA, Los Angeles, California 90095*

October 12, 2006

Abstract

Fault tolerance is an efficient approach adopted to avoid or reduce the damage of a system failure. In this work we present the results of a fault injection campaign we conducted on the Duplex Framework (DF). The DF is a software developed by the UCLA group [1, 2] that uses a fault tolerant approach and allows to run two replicas of the same process on two different nodes of a commercial off-the-shelf (COTS) computer cluster. A third process running on a different node, constantly monitors the results computed by the two replicas, and eventually restarts the two replica processes if an inconsistency in their computation is detected. This approach is very cost efficient and can be adopted to control processes on spacecrafts where the fault rate produced by cosmic rays is not very high. In order to test the reliability of the DF we wrote a simple fault injector that injects faults in the virtual memory of one of the replica process, causing the process to crash or produce erroneous outputs. For this purpose we used two different applications, one that computes an encryption of a input file using the RSA algorithm, and another that optimizes the trade-off between time spent and the fuel consumption for a low-thrust orbit transfer, but with little modification of the original code, any application written in C or Fortran, could be used. Preliminary results show the potential of such approach in recovering from system failures.

1 Introduction

Dependability and safety are a source of great concern in today's world, where many real-time systems are mainly controlled by software with little or no human supervision. In systems such as a nuclear power facility, a missile, a satellite, an aircraft and many others, dependability and safety are mandatory requirements. It is known that no matter how cautious one is in designing, fabricating, assembling the components, and writing the necessary software for a system, there is always the possibility that

a component may fail or a bug may go undetected and manifest itself in the operational phase causing a fault in the system. In order to reduce the probability of such an occurrence, and to contain the damage that it could eventually cause, two main approaches are normally adopted: fault prevention and fault tolerance.

Fault prevention normally consists of two stages: fault avoidance and fault removal. Fault avoidance, as the name suggests, tries to avoid the presence of faults in the system. At a hardware level this is achieved by using reliable, and often expensive, components. Rad-hard components are frequently used in spacecraft, because they are highly immune to upsets caused by cosmic rays. These types of components have much lower performance than similar commercial parts,

Another precaution taken to avoid faults is to adopt very conservative methods in designing and assembling the system, which might cause it to be inflexible and of a limited capability. In spite of all the precautions being taken, a fault may still appear at some point, due to an hardware failure or for an undetected bug in the software code.

Fault removal consists in finding the error and then removing it. Obviously this is not always successful. A spacecraft could be unreachable, or there may be no time to fix the problem, for instance if a component fails during an emergency procedure.

Fault tolerance is another way of tackling the problem. The main idea behind this approach is to accept the possibility that a system failure may occur even after all attempts to avoid it, and then to use a procedure that allows an automatic system recovery in an acceptable amount of time. One way of doing this is by duplicating the system or critical parts of it.

With two copies of the same hardware and software, one can detect a hardware error by comparing the results obtained by both - if the two results differ. However, with only two software duplicates running it is only possible to detect the anomalous behavior, without knowing which one has produced the wrong result. By regularly checkpointing the system, i.e. saving its status, it is possible to restore the two processes from the last checkpoint at which they both provided agreeing results if the error was transient in nature. (Permanent faults cause continued disagreement, and must be resolved by diagnostic tests). Using three, or in general N , copies of the same program (Triple or N Modular Redundancy, TMR or NMR, respectively) to compute the same task, one can assume that the probability of having two or more copies to fail and compute the same erroneous results is very close to zero. By applying majority-voting, one can not only detect possible mistakes in the computation, but also obtain the correct results. In this case the program producing the wrong result can be restarted and updated with the status of one of the programs that provided the right result. The main advantage of this approach is the fact that it is possible to use commercial off-the-shelf (COTS) components, greatly reducing the cost of the system. Moreover, since COTS are normally more powerful and versatile than rad-hard components, the performance of the system is highly improved.

In this work we are going to present the results of a fault injection campaign, where we injected faults into an application running on our Duplex Framework (DF), in order to test its performance and functionality and its ability to recover from a fault. The DF is a software environment developed by the UCLA group [1, 2] with tools that are designed to run single-process applications in duplex, i.e. to run two duplicates

of the same executable on two different nodes of a processor cluster. A third node in the cluster provides access to shared file system and it provides a "comparator service" that compares the results outputted by the two replica processes and, if any difference is detected, it calls for restarting the two jobs. The third node is intended to emulate a spacecraft data storage device. It is not susceptible to transient errors since a spacecraft data storage system is likely to use error detecting and correcting codes and to be implemented with rad-hard controllers.

This system is implemented on a testbed that uses the UCLA-developed fault-tolerant Cluster Manager Middleware (CMM) that schedules programs coordinates and synchronizes the operations of the cluster [1, 2]. The CMM testbed has a core functionality that is triplicated on three processing nodes and that can maintain the cluster's operation even if a node fails or crashes. It can relocate itself and restart any application process on the remaining resources.

Section 2 describes the experimental setup and its components such as the DF and the UCLA CMM. Section 3 describes the design of a simple fault injector program that was developed to conduct the fault injection campaign. In Section 4 we describe the applications used for the experiment and how we proceeded in performing it while in Section 5 we present the results obtained from the campaign. Section 6 is dedicated to some comments and conclusion we inferred from the experiment.

2 Experimental Setup and Components

The DF is a software that is able to run a single-process application in duplex. Theoretically it should be possible to take any code written in C or in FORTRAN and run it on this framework, as long as the code is compiled with the scripts provided. The main idea behind the DF is to have two duplicates of the same software running on two different nodes of a COTS cluster, and then another process running on a third one which plays the role of a smart spacecraft data storage system. It provides access to shared bulk storage, supports user-based atomic checkpointing that is implemented within the duplicated application process, and it provides a comparator function (which is assumed to be fault free). The comparator function is in charge of comparing and eventually detecting differences in the results computed by the duplicates. The comparator uses a File Server Library (FSL) which allows it to monitor all the data input or output by the two replica processes anytime they perform a series of I/O operations. This is achieved by renaming, at compile time, the symbols *open*, *close*, *lseek*, *read(v)*, *write(v)* and *printf* and the renamed symbols are then handled by the FSL. Every time one of the above I/O operations is performed by the two replicas, the data is redirected to the comparator. In order to save bandwidth one replica sends the actual message, while the other sends a "digest", i.e. a shorter message containing the checksum of the data being sent. The comparator computes the checksum on the message received from one replica and compares it with the digest obtained from the other. If the two results differ, the comparator assumes that one of the two replicas has provided a faulty result and asks the CMM to restarts the replica processes. The comparator also uses timers that are supposed to be reset by the replicas. These are used to detect if one of the two duplicated processes unexpectedly crashed or was delayed an unacceptable length of

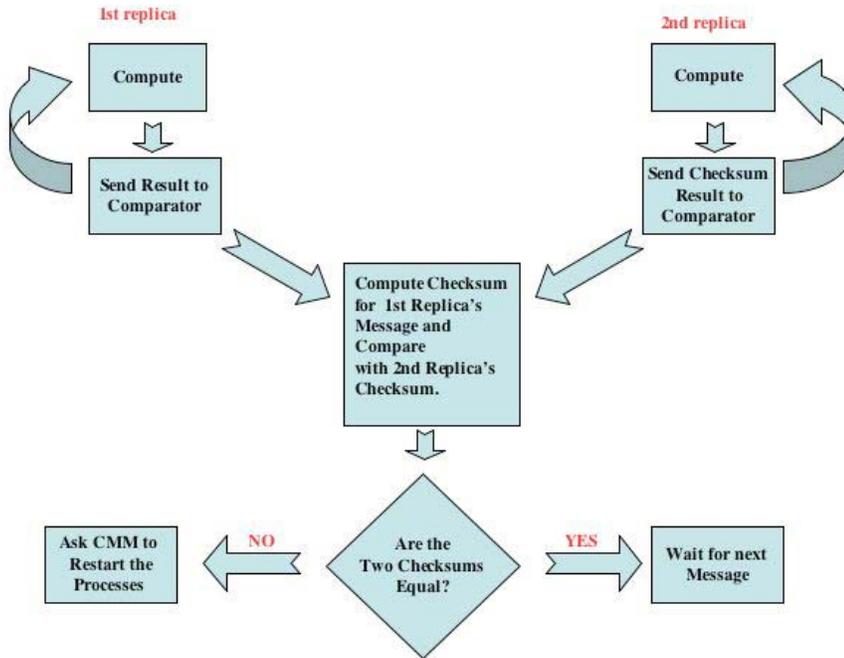


Figure 1: *Modus Operandi* of the DF.

time. If a replica fails to reset a timer within a designated interval, it assumes that the process crashed and it alerts the CMM and requests that it restart the two processes. Fig. 2 explains diagrammatically how the DF works.

2.1 The Cluster Management Middleware (CMM)

The DF lies above the CMM. The CMM is a middleware layer that supplies application software with fault-tolerant control and fault management on a LINUX-based processor cluster. It was developed at UCLA with support from the JPL/NASA Remote Exploration Experimentation (REE) project. The intent of REE was to use COTS hardware and software to deploy scalable supercomputing technology in space, where the error rate is much larger than on earth due to the presence of cosmic rays and high energy protons.

The most common approach used to protect spacecraft computer from these errors is to use rad-hard instrumentation. Rad-hard processors have much lower performance than equivalent COTS parts. Thus the REE approach was to use a COTS-base fault-tolerant cluster, which is able to detect and recover from faults without compromising

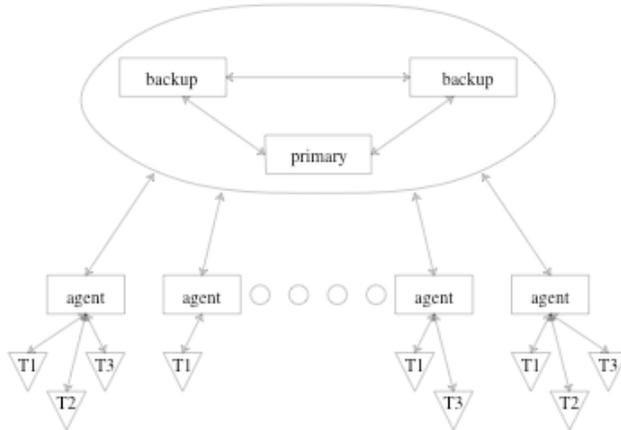


Figure 2: The structure of the CMM

the functionality of the system.

The CMM implements a fault-tolerant, triply modular redundant (TMR) Cluster Manager (CM) running on three of the nodes and a set of agents (one for each node) that allows the CM to control, monitor, and provide fault-tolerance support to the applications running on the cluster (see Fig. 2).

The CMM is in charge of scheduling, allocating resources, coordination of system level monitoring and fault recovering procedures and they all compute the same operations. The agents communicates with the managers reporting the status of the nodes and allowing them to control the system. Indeed, through the managers, the agents can receive the command to start, kill and schedule user application, and even to start a new manager replica if one of the three fails. The manager group consists in a primary replica and two backup replicas. Each agents communicates with the managers through the primary replica, which receives each message, numbers it, and then forwards it to the backups. This procedure guaranties that the messages received by the manager group are always in sequence. On the other hand the managers send their own messages directly to each agent allowing the agents to vote out a disagreeing manager. All the communication among the managers and between the managers and the agents is authenticated or signed [3]. If one agent receives conflicting messages, it immediately reports the discrepancy to the manager group which starts a self-diagnostic procedure where their internal status is compared. If a faulty replica is determined, the process is terminated and a new replica restarted and updated with the status of one on the functioning replicas.

2.2 Cluster Specifications

The cluster on which we conducted the fault injection campaign is composed of:

- six 450 MHz Pentium III with 512 MB of memory running a fully patched version

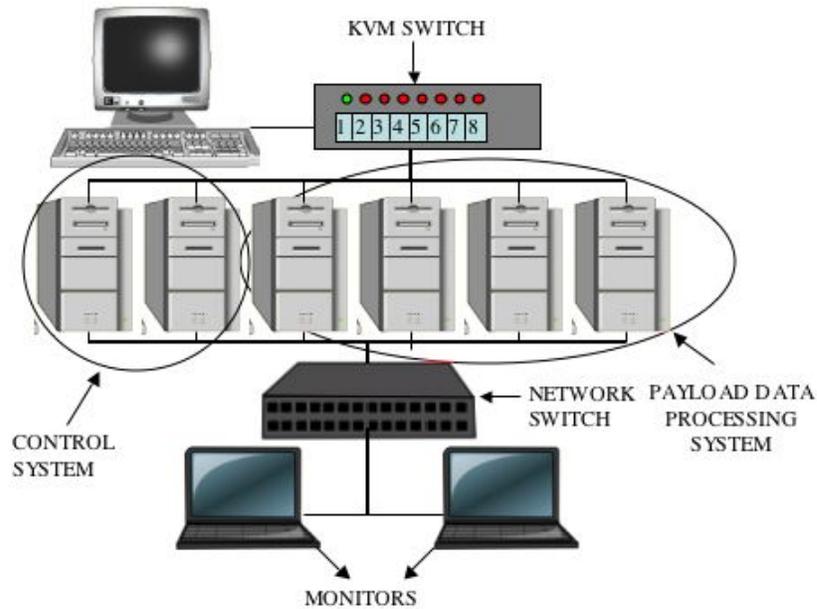


Figure 3: Cluster setup

of Linux Red Hat 9

- two G4 Power PC (PCC)
- a KVM switch
- a network switch Super Stack II Switch 3300.

Four of the six PCs (Payload Data Processing System) are used as compute nodes where all the heavy computation is performed while the other two are used as control system. They are used as interface between the operator and the CMM, allowing to start the CMM, running processes on the DF and checking the system status. Having two PCs as control system is something inherited from the design of the REE project, but it can be restructured such that one is sufficient. The two G4 PCCs are used as monitors. One displays the status of the cluster, such as the nodes on which the managers or the different applications are running, and their process ID. The other monitor displays the output that has been redirected by the File Server Library (see section 2). The six PCs and the two PCCs are all connected through a network switch in a logical star topology with one of the control system PC at the center (the physical topology is still a star but with the switch at the center). Finally a KVM switch allows to have one monitor and one keyboard to control all the six nodes (see Fig 2.2).

```

00310000-00325000 r-xp 00000000 08:07 180292 /lib/ld-2.3.2.so # text
00325000-00326000 rw-p 00015000 08:07 180292 /lib/ld-2.3.2.so # data
00d1b000-00e4e000 r-xp 00000000 08:07 98372 /lib/tls/libc-2.3.2.so # text
00e4e000-00e51000 rw-p 00132000 08:07 98372 /lib/tls/libc-2.3.2.so # data
00e51000-00e54000 rw-p 00000000 00:00 0 # BSS
08048000-0804c000 r-xp 00000000 08:07 229410 /bin/cat # text
0804c000-0804d000 rw-p 00003000 08:07 229410 /bin/cat # data
0823d000-0825e000 rw-p 00000000 00:00 0 # BSS
bfff7000-c0000000 rw-p fffffb000 00:00 0 # stack

```

Figure 4: Typical result of a *maps* system call.

3 The Fault Injector

As mentioned in section 2, the DF is able to run a given application in duplex. In order to test the ability of the framework to restart the replica processes in case one of the two provides an erroneous result or if it unexpectedly terminates, we have developed a simple fault injector with the purpose of generating faults in one of the two replicas. The injector is run as a thread which is spawned from the main process in which we desire to inject faults. The main steps performed by the injector are

- to obtain the segments of the virtual memory areas (VMA) of the running process from the */proc/ <pid> /maps* system call
- to generate a random memory location within the range of address previously obtained
- to gain writable access to the desired memory location using the *mprotect()* system call
- to change the content of the corresponding memory location.

In the first step the thread utilizes the PID to get information about the segment structure of the VMA. Fig. 3 shows the typical results of such a system call. The first two columns show the beginning and the ending address of a specific segment in the virtual memory. The third column describes the permission of that area (r=read, w=write, x=execute). The other columns are not important for our discussion. In this case we showed the VMA for the "cat" command. The */bin/cat* command has a text segment, where the executable is loaded, a data segment where the initialized variables reside, a "block started by symbol" (BSS) segment where the uninitialized variables are. The cat commands uses some libraries, each one with its own set of segments. Finally there is the stack segment of the process. The gap between the BSS of */bin/cat/* and the text segment of the linked library (or between the BSS of the library and the stack) allows the application to dynamically allocate memory on the heap.

Once the memory structure has been determined, we randomly generate an address that falls in one of the previous segments. Since not all segments have write permission, we need to modify the privilege access to the area of memory of interest. For this purpose we use the system call *mprotect* (`int mprotect(void * address, size_t length,`

int permission)), which allows to change the permission of an area starting from a specified address and ending at address plus length (length has to be a multiple of the VMA page size). Once we gain writable access, we change the value at that address (as a matter of fact we write an integer value at that address, modifying the next four bites on our 32 bit system), and then we restore the original permission.

Most of the injected faults have no effect, because the modified location is unused or will be written over before the next use. These are called "ineffective" faults. Typically, the "effective" faults caused by the injector, are seen to be

- segmentation fault, if we write in a location out of bounds,
- illegal instruction, if the modified area belongs to the code segment and it is executed,
- modification of data, if a data segment location has been modified.

The first two types of faults cause the program to crash, while the third one might either cause the program to crash or to produce erroneous results.

The fault injector performs injections at a constant rate, but one injection does not necessarily generate a fault. Since the injection rate is constant and the location is randomly picked, one expects the probability distribution of faults in time to follow a Poisson distribution of type

$$P(t) = Ae^{-\lambda t} \tag{1}$$

where λ is the probability per unit time. We performed a simple test to show that indeed our injector follows such a distribution. We injected in a test application at a constant rate of 5 *msec* and then we counted how many faults occurred in each time interval of 4.5 seconds. The result shown in Fig. 5 is for a total of more than 14000 faults. The green curve is the result of a best fit, where the points are fitted to the distribution in Eq. 1. As one can see that the distribution is Poisson-like.

4 Application Descriptions and Debugging the Experiment

The purpose of our work is to test the DF and check its ability to recover from erroneous results. As described previously (see section 2) the DF runs a single-process application on two nodes of a cluster and a comparator application on a third node that is in charge of comparing the results of some I/O operations performed by the two replica processes. In any difference in the results, independently computed by the two replicas, is detected, then the comparator asks the group manager to restart the two processes. In order to test the DF we used two different applications, one provided by the same group (the UCLA group) that developed the framework (that we will call `duplex_test_app` from now on), and one provided by two of the authors in Ref. [4] (`ga_qlaw` from now on). We first ran the test applications on the DF without performing any injection and recording the result at the end (the final encrypted file, for the `duplex_test_app` or the fuel consumption and the time spent, for the `ga_qlaw`). We considered this result as the exact outcome of the computation. We then ran two separate versions of a given test application on the DF. On one of the replicas we ran a modified version of the

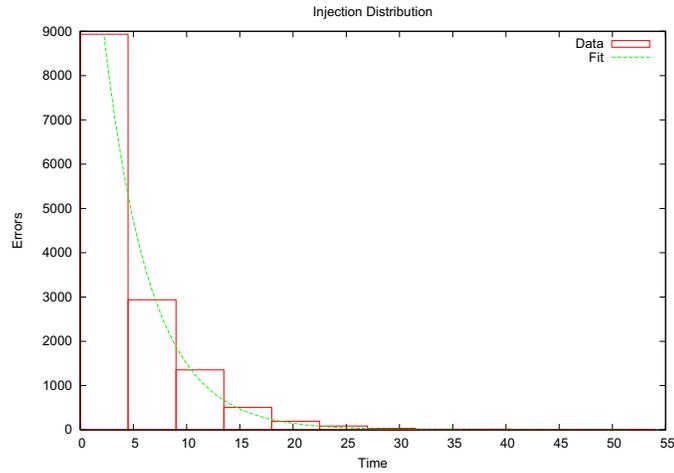


Figure 5: Error probability distribution generated by the injector.

code in which we added the fault injector thread, while on the remaining one we ran the untouched version. We then recorded the final result and compared it with the exact one previously computed.

4.1 The duplex_text_app.

The duplex_test_app is a program that encrypts an input message using the RSA algorithm for a given number of times and writes the final result into a file. This application allows also to do "checkpointing", i.e. to save the status of some variables needed for the computation of the next encryption. Checkpoints allow to use the rollback procedure, i.e. if the program is restarted due to a crash or an inconsistency detected in the computation, it is not necessary to recompute everything from scratch. Instead one could use the previously saved variables and start the computation using the results from the last checkpoint. The rollback procedure is not implemented in the DF, but needs to be implemented in the user application. The UCLA group provides a library with a set of calls that can be helpful to implement such a procedure.

The replica processes compute the encryption of the input file, and write the partial result regularly into a temporary file. The data are then redirected and compared by the comparator, which will eventually restart the process if a mismatch in the digest is found.

4.2 The `ga_qlaw`.

The second application we used to test the DF is a code written to optimize the trade-off between fuel consumption and time spent for a low-thrust orbit transfer. The optimization is reached by using a multi-objective genetic algorithm. Without going into the details on how the software operates, the objective is reached by computing different iterations, called generations, and trying to improve the quality of the solution at each step.

In order to add the rollback feature to the program, it is necessary to slightly modify the code. At each generation the variables necessary to compute the next iteration are saved as a checkpoint. To provide atomicity, these output files are double-buffered, i.e. they are saved alternatively into two different files. A status variable, saved into a different file, determines in which file the last checkpoint was saved correctly and therefore from which file the rollback needs to be done, providing a "commit" action. Partial results computed at each generation are written into a file and sent to the comparator, that will take the appropriate actions depending on the result of the comparison.

4.3 Debugging

The addition of the injector thread into the main application in which we wanted to inject random faults, posed some problem to the correct operation of the DF. We performed two sets of runs of the `duplex_test_app` for debugging purpose. The first set consisted in 21 runs each of which computed 500 encryptions, while the injection rate was 2 Hz. Only 12 out of 21 runs completed successfully, in the sense that the final result was equal to the one obtained when the application was run without injections. Of the 9 unsuccessful runs, in 4 cases a fault that occurred before the first checkpoint would cause the application to restart, but not from the first encryption, as it would be expected. We manage to avoid this kind of problem (let us call it first type of issue from now on) by starting the injections after the first checkpoint. In the second set of runs, where this precaution was taken, this inconvenient was not present anymore. It seemed that the problem was due to a bug in the rollback procedure implemented in the `duplex_test_app` and not an issue related to the DF itself, in fact this kind of issue was not present in the `ga_qlaw` application runs, in which we implemented our own checkpoint and rollback algorithm.

In the other 4 unsuccessful runs, the application would compute all the 500 encryptions and write the final result to a file. At this point there would be a sequence of restarts and crashes that would eventually end. The process would then rewrite the final result into a file, that turned out to be incorrect (second type of issue from now on). In the last failed run the comparator, that was assumed to be fault free, unexpectedly died (third type of issue).

We performed a second set of 17 runs (1000 encryptions and 1 Hz injection rate) in which the injections started after the first checkpoint. Ten runs were successful, while of the 7 unsuccessful runs, 4 were of the second type and 3 of the third type and none of the first type.

The second type of issue turned out to be also related to the implementation of the checkpointing. If an effective fault occurred after writing the final result, but before

finalizing the process, the program would restart from the last checkpoint performed instead of simply finalizing end ending. The rollback did not have a status variable indicating the end of the process, but only the last successful checkpoint computed. We modified slightly the code adding such status variable that is set once the process has finished the computation. In case of a restart after writing the final result, the program would first check this status variable and if set, it would end the process without modifying the result previously computed.

At the time when the experiment was performed we were not able to identify the exact cause of the problem. To avoid the issue we decided to interrupt the injections before the last encryption and then voluntarily kill the process. The comparator would then restart the replica processes, but the injector thread was not spawned.

As we will see in Section 5 after taking these two precautions, i.e. starting the injections after the first checkpoint and interrupting them before the last one, in the third set of runs neither first nor second type of issue were present. The reason of the third type of issue remains unclear.

Also the `ga_qlaw` application presented some issue during the MPI finalization of the process. In some instances, after the application reached the end and the result was written, an MPI error would occur. The injected replica would finish while the non injected replica and the comparator kept running. The final result written by the injected replica was correct. Although we could not pinpoint the origin of the problem, we succeeded in eliminating the issue by stopping the injection before the last generation was computed. The fact that the final result was correctly computed showed that the DF worked as expected as far as detecting inconsistency is concerned.

5 Injection Campaign Results

After the debugging procedure, we performed a set of 30 runs of the `duplex_test_app` computing 300 encryptions at 2 Hz injection rate. There was only one unsuccessful run in which the application crashed.

We then ran the `ga_qlaw` application for a small number of generations (the run times were of the order of few tens of minutes). The first set consisted in a total of 50 runs computing 40 generations at an injection rate of 1 Hz, while in the second set we performed 30 runs computing 15 generations at an injection rate of 0.5 Hz. In the first set the fuel consumption and the time spent were computed once per generation, while in the second set were computed six time.

To test the functionality of the DF over a longer run time, we also performed a set of 10 runs of the `ga_qlaw` application in which the approximate run time was between 15 and 16 hours. The total number of generations was 350 and the time between two injections varied from 7.5 to 30 seconds at step of 2.5 seconds (the injection rate varied from 0.033 to 0.133 Hz). In Figs. 6, 7 and 8 we show the plots of the number of injections, number of restarts (each restart correspond to a fault that caused an error) and the effectiveness (defined as the ratio of the number of restarts and the number of injections divided by a hundred) as a function of the injection rate (expressed in faults/min), for the long runs. We also plotted in Fig. 9 the runs distribution as a function of the effectiveness, which shows that in average we had one effective fault

Application Name	Fault Injection Rate (Hz)	Total # of Runs	# Successful Runs	# Unsuccessful Runs
duplex_test_app I	2	21	12	9
duplex_test_app II	1	17	10	7
duplex_test_app III	2	33	32	1
ga_qlaw I	1	50	50	0
ga_qlaw II	0.5	30	30	0

Table 1: Summary of the results of the short time runs.

Application Name	Average # Errors per Run	Average # Errors per Run per Hour	Average # Injections	Average Effectiveness
duplex_test_app I	33.3	64.1	3698	0.89
duplex_test_app II	19.3	32.8	2123	0.91
duplex_test_app III	10.6	61.2	1239	0.85
ga_qlaw I	9.74	36.4	921	1.01
ga_qlaw II	8.13	24.3	587	1.35

Table 2: Summary of the results of the short time runs.

every one hundred injections.

We summarize the results of the entire campaign in Tabs. 1, 2 and 3 including the successful runs performed during the debugging phase. All the averages computed refer only to the successful runs. In Tabs. 1 and 2 we report the results for the three duplex_test_app runs and the two short runs of the ga_qlaw application. The different columns in Tab. 1 are, respectively, the application name, the fault injection rate, the total number of runs, the number of successful runs and the number of unsuccessful runs, while in Tab. 2 the columns are application name, the average number of errors per run, the average number of errors per run per hour, the average number of injections per run and the average effectiveness per run. In Tab. 3 we report the results for the long runs of the ga_qlaw application. The first columns is the injection rate, the second the outcome of the run (Successful/Unsuccessful), then the number of errors for that run and finally the number of errors per hour for that specific run. Must be noted, that in all runs that reached the completion of the task, despite the number of errors injected and the number of restarts, the final result was always correct. The only unsuccessful run was due to an unexpected crash of the comparator application.

Based on the numbers in Tabs. 1, 2 and 3 we can compute a lower limit on p , the probability of failure of the DF. Given that for a total of $N = 2075$ errors, all of them were detected and the DF never failed, we have $p < 1/N \pm 1/\sqrt{N}$, yielding a reliability lower bound for the DF $R = 1 - p < 1/N \pm 1/\sqrt{N} = 0.99952 \pm 0.02195$. This is obviously a very conservative number, which could be improved by running more

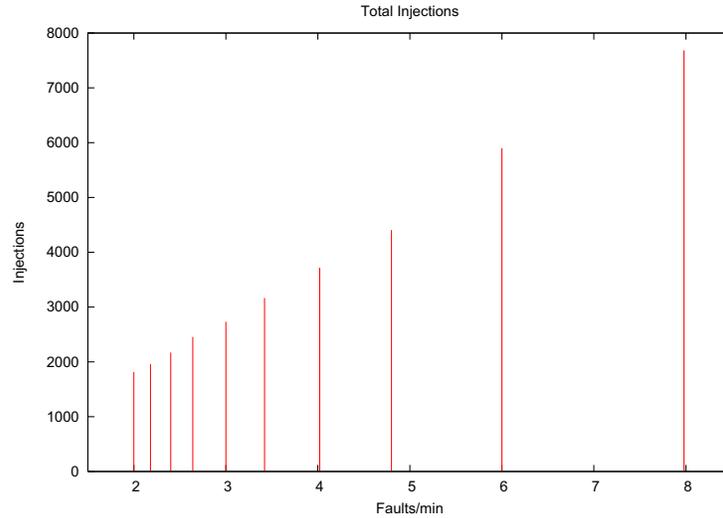


Figure 6: Total injections for long run as a function of the injection rate.

tests.

6 Comments and Conclusions

6.1 Comments

As mentioned before, in some occasion the DF or the CMM presented some unexpected behavior that needs more investigation that we summarize as follow:

- In few occasions the comparator, that was assumed to be fault free and therefore no injections were made in the corresponding application, crashed. The injected replica eventually died because of a fault, while the other replica kept running.
- In several occasions, if a fault would occur in the `duplex_test_app` before the first checkpoint, the replica processes would not be restarted from the beginning, but from an arbitrary checkpoint, providing at the end an erroneous result. The problem was not due to the DF, but to the rollback algorithm implemented in the test application.
- In few instances in the `duplex_test_app`, after the two replicas had reached the total number of encryptions and exited, the comparator would restart the processes several times before exiting the application. The result output in this case was wrong. Also in this case the reason of the malfunction was ascribable to a problem in the rollback procedure.

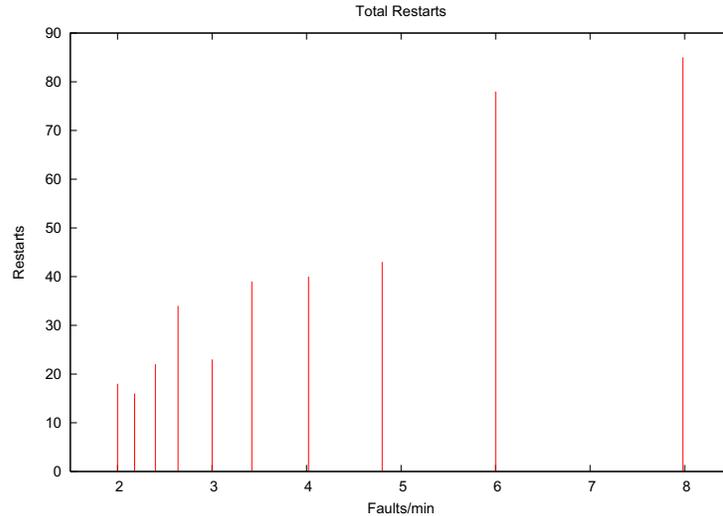


Figure 7: Total restarts for long run as a function of the injection rate.

- Occasionally, in the `ga_qlaw` application, after the last generation was computed, a MPI error would occur. In this case the injected replica would exit giving the correct result, while the comparator and the fault free replica would still run. This problem was solved by stopping the injection before the last generation was computed.
- In the CMM, after hours of running, the manager applications would unexpectedly die.

Except in few cases where the comparator unexpectedly terminated, most of the anomalous events we just described, specifically the ones that took place during the run of the `duplex_test_app`, were ascribable to a problem in the implementation of the rollback procedure and not due to the DF itself.

The problem in the `ga_qlaw` application showed that the DF worked correctly providing the expected result, but the CMM failed in finalizing the processes.

In the few cases in which the comparator died unexpectedly we were not able to track the origin of the issue. The last two issues require more studying to pinpoint the origin of the problem.

An important point is how difficult is and how much modification an application would require in order to be run on the DF. We took other two simple C language written applications and compiled on the DF. The number of modifications in the Makefile and the code itself were quite limited, and did not present a particular issue. The main modifications required are due to the fact that all the inputs can not be done from a terminal, but need to be read from an input file. Moreover, the use of

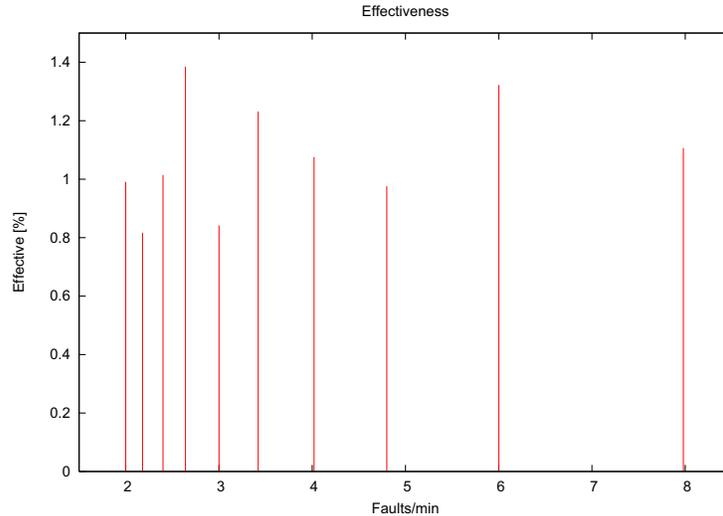


Figure 8: Effectiveness for long run.

random number generator could be a source of problem. If for instance the time is used to seed it, if the two replicas are not synchronized (which is quite like to be the case) the sequence of numbers generated will be different and consequently the results produced by the two replicas may differ, causing the comparator to restart the processes each time. Both applications ran correctly. We also manually killed one of the replica processes during the computation. In all cases the comparator, by means of timer, detected that one of the processes terminated and requested the CMM to restart the application.

6.2 Conclusion

We run a fault tolerant campaign in order to test the DF. The main goal of the campaign was to show the potential of the approach adopted by the DF to guaranty reliability in the execution of a process. We have shown the ability of the DF to restart a process when a crash occurred or an inconsistent computation was performed. An important result is that during the entire campaign we never witnesses a case in which the two replicas finished the computation, but they provided disagreeing results. From the data collected during the experiment we provided a very conservative estimate for the reliability of the DF ($R = 0.99952 \pm 0.02195$) that could be easily improved by running more tests.

We have however noticed some unexpected behaviors such as the crash of the comparator, or problems experienced during the finalization of the the processes. All these

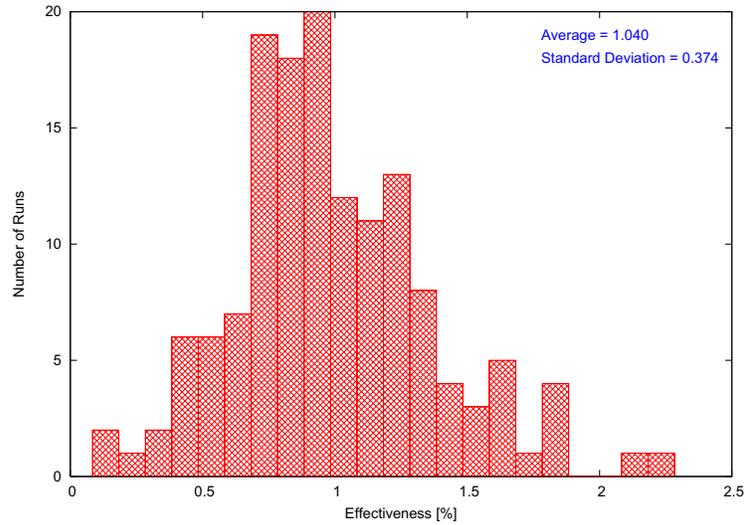


Figure 9: Runs distribution as a function of effectiveness

issues require more testing, upgrading and debugging, but overall the DF behaved as designed.

We believe that the main advantage of the philosophy adopted to implement the DF is the possibility of using COTS components in building the computing nodes of the cluster where to run it. Such approach could, for instance, highly reduce the cost of the onboard instrumentation on a spacecraft and greatly enhance the performance of the system. Rad-hard components have the disadvantage of not being particularly powerful and moreover, the rate at which these components are improved is very small compared to equivalent commercial components. One could use a rad-hard processor to perform vital tasks onboard, such as schedule events and monitor the system, but delegate the heavy computational part to the COTS nodes. This opens a all new scenario on the type of computation that can be performed onboard, such as realtime image processing or even being able to conduct simple experiment on material samples extracted from the soil of a planet.

More effort has to be put in promoting and testing this fault tolerant approach in developing software in order to improve its reliability and make it ready for flight missions.

References

- [1] M. Li *et al*, *Proceedings of International Conference on Parallel and Distributed Computing and Systems*, Anaheim, CA, pp. 480-485, August 2001.
- [2] D Goldberg *et al*, Computer Science Department Technical Report CSD-010040, University of Los Angeles, CA.
- [3] L. Lamport *et al*, *ACM Transactions on Programming Languages and Systems*, 4(3), pp. 382-401 (July 1982).
- [4] S. Lee *et al*, AAS 05-392 Paper, AAS/AIAA Astrodynamics Specialist Conference, Lake Tahoe, August 8-11, 2005.

Inj. Rate. (Hz)	Result (S/U)	Num. Err.	Num. Err. /h
0.033	S	18	1.2
0.036	S	16	1.1
0.040	S	22	1.5
0.044	S	34	2.2
0.050	S	23	1.5
0.057	S	39	2.5
0.067	S	40	2.6
0.080	S	43	2.8
0.100	S	78	4.8
0.133	S	85	5.3

Table 3: Summary of the results of the long time runs for the ga_qlaw application.