On the Requirements of New Software Development

Vincenzo De Florio and Chris Blondia
University of Antwerp, Department of Mathematics and Computer Science
Performance Analysis of Telecommunication Systems group
Middelheimlaan 1, 2020 Antwerp, Belgium
Interdisciplinary institute for BroadBand Technology
Gaston Crommenlaan 8, 9050 Ghent-Ledeberg, Belgium

Abstract

Changes, they use to say, are the only constant in life. Everything changes rapidly around us, and more and more key to survival is the ability to rapidly adapt to changes. This consideration applies to many aspects of our lives. Strangely enough, this nearly self-evident truth is not always considered by software engineers with the seriousness that it calls for: The assumptions we draw for our systems often do not take into due account that e.g., the run-time environments, the operational conditions, or the available resources will vary. Software is especially vulnerable to this threat, and with today's software-dominated systems controlling crucial services in nuclear plants, airborne equipments, health care systems and so forth, it becomes clear how this situation may potentially lead to catastrophes. This paper discusses this problem and defines some of the requirements towards its effective solution, which we call "New Software Development" as a software equivalent of the well-known concept of New Product Development. The paper also introduces and discusses a practical example of a software tool designed taking those requirements into account—an adaptive data integrity provision in which the degree of redundancy is not fixed once and for all at design time, but rather it changes dynamically with respect to the disturbances experienced during the run time.

1 Introduction

We live in a society of software-predominant systems. Computer systems are everywhere around us—from supercomputers to tiny embedded systems—and we all know how important it is that services appointed to computers be reliable, safe, secure, and flexible. What is often overlooked by many is the fact that most of the logics behind those services, which support and sustain our societies, is in the software layers. Software has become the point of accumulation of a large amount of complexity. Software is ubiquitous, it is mobile, it has pervaded all aspects of our lives. Even more than that, the *role* appointed to software has become crucial, as it is ever more often deployed so as to fulfill mission critical tasks. How was this possible in such a short time? The key ingredient to achieve this change in complexity and role has been the conception of tools to manage the structure of software so as to divide and conquer its complexity.

- *Dividing* complexity was achieved through specialization, by partitioning it into system layers (for instance the OS, the middleware, the network, the application, and so forth).
- *Conquering* that complexity was mainly reached by hiding it by means of clever organizations (for instance through object orientation and, more recently, aspect and service orientation).

Unfortunately, though made transparent, still this complexity is part of the overall system being developed. As a result, we have been given tools to compose complex software-intensive systems in a relatively short amount of time, but the resulting systems are often entities whose structure is unknown and are likely to be inefficient and even error-prone.

An example of this situation is given by the network software layers: We successfully decomposed the complexity of our telecommunication services into well-defined layers, each of which is specialized on a given sub-service, e.g. routing or logical link control. This worked out nicely when Internet was fixed. Now that Internet is becoming predominantly mobile, those telecommunication services require complex maintenance and prove to be inadequate and inefficient. Events such as network partitioning become the rule, not the exception [2]. This means that the system and fault assumptions on which the original telecommunication services had been designed, which were considered as permanently valid and hence hidden and hardwired throughout the system layers, are not valid anymore in this new context. Retrieving and exploiting this "hidden intelligence" [9] is very difficult, which explains the many research efforts being devoted world-wide to cross-layer optimization strategies and architectures.

Societal bodies such as enterprises or even governments have followed an evolutionary path similar to that of software systems. Organizations such as enterprises or even governments have been enabled by technology so as to deal with enormous amounts of data; still, like for software systems, they evolved by trading ever increasing performance with ever more pronounced information hiding. The net result in both cases is the same: Inefficiency and error-proneness. This is no surprise, as system-wide information that would enable efficient use of resources and exploitation of economies of resources is scattered into a number of separated entities with insufficient or no communication flow among each other. This is true throughout the personnel hierarchy, up to the top: Even top managers nowadays only focus on fragmented and limited "information slices". Specialization (useful to partition complexity) rules, as it allows an enterprise to become more complex and deal with a wider market. It allows unprecedented market opportunities to be caught, hence it is considered as a panacea; but when the enterprise is observed a little closer, often we observe deficiencies. In a sense, we often find out that the enterprise looks like an aqueduct that for the time being serves successfully its purpose, but loses most of its water due to leakage in its pipelines. This leakage is often leakage of structural information—a hidden intelligence about an entity's intimate structure that once lost forbids any "cross-layer" exploitation. Consequently efficiency goes down and the system (be it an enterprise, an infrastructure, a municipality, or a state) becomes increasingly vulnerable: It risks to experience failures or looses an important property with reference to competitiveness, i.e., agility, that we define here as an entity's ability to reconfigure itself so as to maximize its ability to survive and catch new opportunities. For business entities, a component of agility is the ability to reduce time-to-market. For software systems, this agility includes adaptability, maintainability, and reconfigurability—that is, adaptive fault-tolerance support. We are convinced that this property will be recognized in the future as a key requirement for effective software development—the software equivalent of the business and engineering concept of New Product Development [18]. The tool described in this paper—an adaptive data integrity provision—provides a practical example of this vision of a "New Software Development."

The structure of this paper is as follows: Section 2 introduces the problem of adaptive redundancy and data integrity. Section 3 is a brief discussion on the available data integrity provisions. A description of our tool and design issues are given in Sect. 4. In Sect. 5 we report on an analysis of the performance of our tool. Our conclusions are finally drawn in Sect. 6.

2 Adaptive Redundancy and Data Integrity

A well-known result in information theory by Shannon [21] tells us that, from any channel with known and fixed unreliability, it is possible to set up a more reliable channel by increasing the degree of information redundancy. A similar conclusion holds in the domain of fault-tolerance: Let us consider a computer service that needs to sustain a predefined level of reliability when deployed in a given disturbed environment. Furthermore, let us assume that such environment is characterized by a known and fixed scenario of disturbances—for instance, known and bounded electro-magnetic interference. In such a case, the required reliability can be reached by using some fault-tolerance mechanism and a certain level of redundancy (in time, information, design, or hardware) to cope with the disturbances. An example of this approach is described e.g. in [7], where a stable memory is obtained by using a spatial and temporal redundancy scheme. In both cases, when the channel (respectively, the environment) does not change their characteristics, a precise estimation of the unreliability (respectively, the possible disturbances affecting the run-time environment) can effectively enhance reliability.

Unfortunately, such a precise estimation is often not possible or not meaningful. To show this, let us focus on a particular problem, namely that of data integrity: There, the goal is being able to protect data against memory disruptions and other faults that could make it impossible to access some previously saved data. Furthermore, let us consider a particular solution to that problem, namely redundant data structures. In such case redundancy and voting are used to protect memory from possible transient or permanent corruptions. A common design choice in data integrity through redundant data structures is having a static fault model assumption, such as e.g. "during any mission, up to 3 faults shall affect the replicas", which translates into using a 7-redundant cell to address the worst possible scenario. This brings us to a number of observations:

- First of all, such assumptions are the result of worst-case analyses which often are not a viable solution, especially in the case of embedded and mobile devices: A large consumption of resources would severely hamper the operating autonomy without bringing any tangible added value.
- Moreover, a static fault model implies that our data integrity provisions will have a fixed range of admissible events to address and tolerate. This translates into two risks, namely

  1. overshooting, i.e., over-dimensioning the data integrity provision with respect to the actual threat being experienced, and
  2. undershooting, namely underestimating the threat in view of an economy of resources.

  Note how those two risks turn into a veritable dilemma to the designer: Wrong choices at this point can lead to either unpractical, too costly designs or cheap but vulnerable provisions.
- Another tricky aspect in a static choice of the possible faults to affect our service is the hidden dependence on the target hardware platform. This dependence translates in very different failure semantics. In other words, the way the target hardware components behave in case of failure can vary considerably with the adopted technology and even the particular component. As an example, while yesterday's software was running atop CMOS chips, today the common choice e.g. for airborne applications is SDRAM—because of speed, cost, weight, power and simplicity of design [14]. But CMOS memories mostly experience single bit errors [19], while SDRAM chips are known to be subjected to several classes of "nasty" faults, including so-called "single-event effects" [14], i.e., single faults affecting whole chips. Examples include:

  1. Single-event latchup, a threat that can bring to the loss of all data stored on chip [26].
  2. Single-event upset (SEU), leading to frequent soft errors [28, 27].
  3. Single-event functional interrupt (SEFI), i.e. a special case of SEU that places the device into a test mode, halt, or undefined state. The SEFI halts normal operations, and requires a power reset to recover [10].

  Furthermore the already cited paper [14] remarks how even from lot to lot error and failure rates can vary more than one order of magnitude. In other words, the superior performance of the new generation of memories is paid with a higher instability and a trickier failure semantics.

  Despite these veritable pitfalls, hardware dependence is mostly not taken into account when developing fault-tolerant services e.g. for data integrity. Ideally, hardware dependence and impact on failure semantics should be explicitly available at service design, composition, and deployment times. Tools could then be conceived to highlight risks and assist the designer of software services.
- Still another aspect derives from the choice of the failure semantics: Addressing a weak failure semantics, able to span many failure behaviours, effectively translates in higher reliability—nevertheless,

  1. it *requires* large amounts of extra resources, and therefore implies a high cost penalty, and
  2. it *consumes* those extra resources, which translates into their rapid exhaustion.

  For instance, a well-known result by Lamport et al. [15] sets the minimum level of redundancy required for tolerating Byzantine failures to a value that is greater than the one required for tolerating, e.g., value failures. Using the simplest of the algorithms described in the cited paper, a 4-modular-redundant (4-MR) system can only withstand any *single Byzantine failure*, while the same system may exploit its redundancy to withstand up to three crash faults—though no other kind of fault [20]. In other words:

After the occurrence of a crash fault, a 4-MR system with strict Byzantine failure semantics has exhausted its redundancy and is no more dependable than a non-redundant system supplying the same service, while the crash failure semantics system is able to survive to the occurrence of that and two other crash faults. On the other hand, the latter system, subject to just one Byzantine fault, would fail regardless its redundancy.

We conclude that, for any given level of redundancy, *trading complexity of failure mode against number and type of faults tolerated* may be considered as an important capability for an effective fault-tolerant structure. Dynamic adaptability to different environmental conditions[1] can provide a satisfactory answer to this need, especially when the additional complexity required to manage this feature does not burden or jeopardize the application.

- A final argument is that run-time environments change, often because services are usually mobile but sometimes also because of external events affecting e.g. temperature, radiation, electro-magnetic interference, or cosmic rays. Applications such as a wireless sensor network to assist a fire brigade, or a spaceborne application orbiting around the sun, are *bistable* by nature: They usually operate in either a "normal" state, where the environmental conditions are not particularly demanding, or in a "critical" state, likely to affect the mission's integrity if no special countermeasures are taken. A sudden fire or a solar flare are examples of events requiring to switch to critical state. Also in such cases a static choice would be unpractical, as the nature of faults is meant to change during a same mission.

Our vision to this problem is that fault-tolerant services should be built with architectural and/or structuring techniques able to decompose their complexity, but without hiding the basic hypotheses and assumptions about their target execution environment, the expected fault- and system models, and the failure semantics assumed for the components our services depend on. Our conjecture is that effective solutions to this problem should come by *considering the nature of faults as a dynamic system*, i.e., a system evolving in time, and by expressing the fault model as a function $F(t)$. Consequently, we believe that any fault-tolerance provision that be able to solve the above mentioned flaws should make use of an adaptative feedback loop [24]. In such loop redundancy would be allocated according to the measured values of $F(t)$, obtained by monitoring a set of meaningful environmental variables.

This paper reports on the design of a tool compliant to such model. Our tool allows designers to make use of adaptively redundant data structures with commodity programming languages such as C or Java. Designers using our tool can define redundant data structures where the degree of redundancy is not chosen at design time but changes dynamically with respect to the disturbances experienced at run-time. In other words, our approach attunes the degree of redundancy required to ensure data integrity to the actual faults being experienced by the system and provides an example of adaptive fault-tolerance software provision. Such property allows to reach adaptability, maintainability, and reconfigurability, which we deem as being three fundamental components to fulfill the requirements of "new software development"—the software equivalent of the business and engineering concept of New Product Development [18].

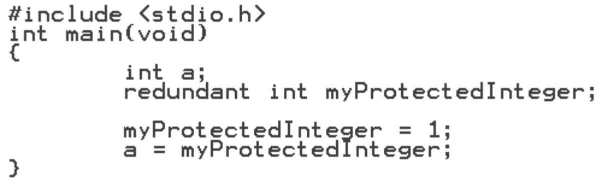
3 A Brief Introduction to Data Integrity Provisions

As well known, redundancy is key to error detection, correction, and recovery [23]. The use of redundancy to detect and correct errors in stored data structures has been investigated for decades. Techniques may be divided into two classes: Those that only support error detection and those that support error correction as well.

Protection regions through codewording [1] and structure marking [25] are examples of error detection techniques. The rationale of such methods is that, when a "wild store" occurs, due to a design or physical fault, or because of a malicious attack, the techniques allow detecting the change[2] and shutting down the service so as to enforce crash failure semantics. This is in general an effective approach, though there exist cases, e.g. unmanned spaceborn missions, where other approaches could be more sensible. One such approach is discussed in the rest of this paper.

[1] The following quote by J. Horning [12] captures very well how relevant may be the role of the environment with respect to achieving the required quality of service: "What is the most often overlooked risk in software engineering? That the environment will do something the designer never anticipated."

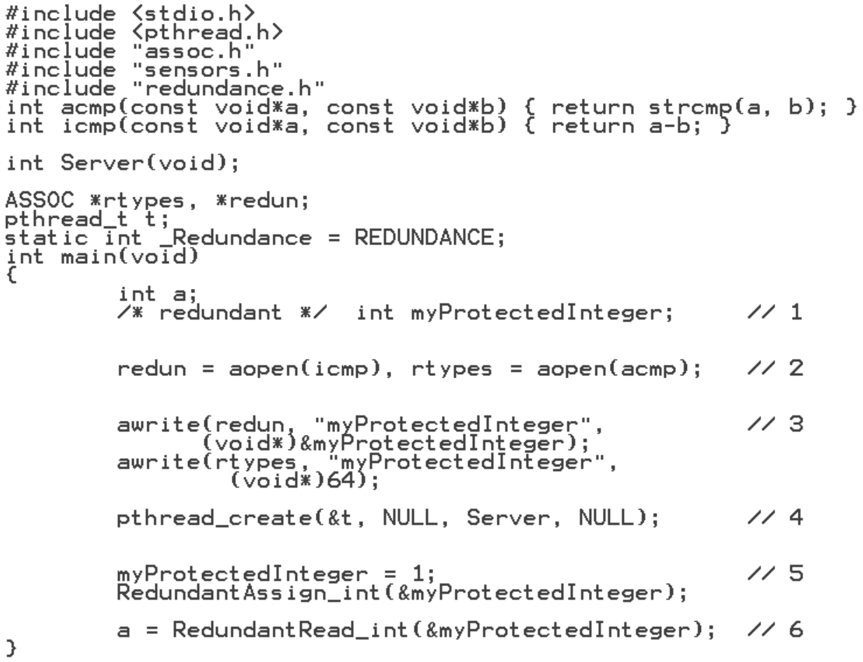
[2] Following [23], we define a *change* as "an elementary modification to the encoded form of a data structure instance."



```
#include <stdio.h>
int main(void)
{
        int a;
        redundant int myProtectedInteger;

        myProtectedInteger = 1;
        a = myProtectedInteger;
}
```

Figure 1. A simple example of use of redundant variables.



```
#include <stdio.h>
#include <pthread.h>
#include "assoc.h"
#include "sensors.h"
#include "redundance.h"
int acmp(const void*a, const void*b) { return strcmp(a, b); }
int icmp(const void*a, const void*b) { return a-b; }

int Server(void);

ASSOC *rtypes, *redun;
pthread_t t;
static int _Redundance = REDUNDANCE;
int main(void)
{
        int a;
        /* redundant */  int myProtectedInteger;     // 1

        redun = aopen(icmp), rtypes = aopen(acmp);   // 2

        awrite(redun, "myProtectedInteger",          // 3
              (void*)&myProtectedInteger);
        awrite(rtypes, "myProtectedInteger",
              (void*)64);

        pthread_create(&t, NULL, Server, NULL);      // 4

        myProtectedInteger = 1;                      // 5
        RedundantAssign_int(&myProtectedInteger);

        a = RedundantRead_int(&myProtectedInteger);  // 6
}
```

Figure 2. An excerpt from the translation of the code in Fig. 1. Variable "_Redundance" represents the current amount of redundancy, initially set to "REDUNDANCE" (that is, 3).

Error correction in general requires more resources in time and space and ranges from corruption recovery in transaction processing systems through redoing [1] to full backup copies. Backup copies can be arranged through bitwise replicas (semantical integrity) or by using redundancy in the representation of the data (structural integrity). Examples of the latter can be found in [23]. An example of the former case is the method described in our paper.

4 Dynamically Redundant Data Structures

Our tool is a translator that loosely parses a source code performing some transformations as reported in the rest of this section. We developed our translator in the C language and with Lex & YACC [17]. The reported version supports the C syntax though the same principles can be easily applied to any other language. Our translator performs a simple task—it allows the programmer to tag scalar and vector variables with a keyword, "redundant," and then instruments the memory accesses to tagged variables. Figure 1 shows how this is done in practice with a very simple example whose translation is provided in Fig. 2. Let us review the resulting code in more detail (please note that item x in the following list refer to lines tagged as "// *x*" in the code):

1. First the translator removes the occurrences of attribute "redundant".
2. Then it performs a few calls to function "aopen" [3]. This is to open the associative arrays "redun" and "rtype". As well known, an associative array generalizes the concept of array so as to allow addressing items

by non-integer indexes. The arguments to "aopen" are functions similar to "strcmp", from the C standard library, which are used to compare index objects. The idea is that these data structures create links between the name of variables and some useful information (see below).

3. There follow a number of "awrites", i.e., we create associations between variables' identifiers and two numbers: the corresponding variables' address and an internal code representing its type and attributes (code 64 means "redundant int").
4. Then the "Server" thread, responsible to allocate replicas and to monitor and adapt to external changes, is spawned.
5. A write access to redundant variable w, of type t, is followed by a call to "RedundantAssign_t(&w)".
6. Finally, reading from redundant variable w, of type t, is translated into a call to function "RedundantRead_t(&w)".

The strategy to allocate replicas is a research topic on its own. Solutions range from naïve simple strategies like allocating replicas into contiguous cells—which makes them vulnerable to burst faults—to more sophisticated strategies where replicas get allocated e.g. in different memory banks, or different memory chips, or even on different processors. Clearly each choice represents a trade-off between robustness and performance penalty. In our current version we separate replicas by strides of variable length.

The core of our tool is given by functions "RedundantAssign_t(&w)" and "RedundantRead_t(&w)", which are automatically generated for each type t through a template-like approach. The former function performs a redundant write, the latter a redundant read plus majority voting. For voting, an approach similar to that in [6] is followed.

What makes our tool different from classical libraries for redundant data structures such as the one in [23] is the fact that in our system the amount of replicas of our data structures changes dynamically with respect to the observed disturbances. We assume that a monitoring tool is available to assess the probability of memory corruptions of the current environment. We also provide an example of such a monitoring tool, which estimates that probability by measuring for each call to "RedundantRead_t" the risk of failure r. Quantity r may be defined for instance as follows: If our current redundancy is $2n+1$, and if the maximum set of agreeing replicas after a "RedundantRead_t" is m, $(1 \leq m \leq 2n+1)$, then

$$r = \begin{cases} (2n+1-m)/n & \text{if } m > n \\ 1 & \text{otherwise.} \end{cases} \tag{1}$$

For instance if redundancy is 7 and $m = 6$, that is if only one replica differs, then $r = 1/3$. Clearly the above choice of r lets risk increase linearly with the number of replicas not in agreement with the majority. Other formulations for r and for the monitoring tool[3] are possible and likely to be more effective than the ones taken here—also a matter for future research.

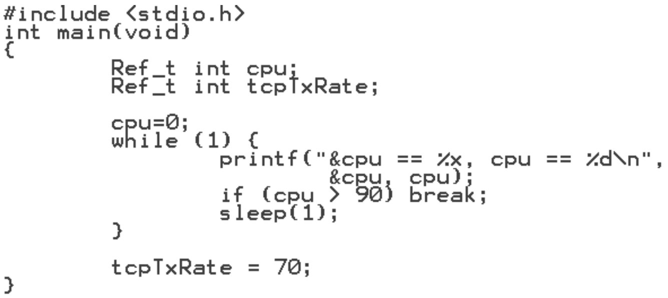
Our strategy to adjust redundancy is also quite simple: If we observe that $r > 0.5$, redundancy is increased by 2; if $r = 0$ for 1000 consecutive calls to "RedundantRead_t", redundancy is decreased by 2. Lower bound and upper bound for redundancy have been set to 3 and 11 respectively.

Each request for changing redundancy is reflected by the "Server" thread into variable "_Redundance" through the scheme introduced in what follows.

4.1 Reflective and Refractive Variables

The idea behind reflective and refractive variables (RR vars) [4] is to use memory access as an abstraction to perform concealed tasks. RR vars are volatile variables whose identifier links them to an external device, such as a sensor, or an RFID, or an actuator. In reflective variables, memory cells get asynchronously updated by service threads that interface those external devices. We use the well-known term "reflection" because those variables in a sense "reflect" the values measured by those devices. In refractive variables, on the contrary, write accesses trigger a request to update an external parameter, such as the data rate of the local TCP protocol entity or the amount of redundancy to be used in transmissions. We use to say that write accesses "refract" (that is, get redirected [13]) onto corresponding external devices.

[3] It is clear that a proactive approach would be more effective than the one reported here, which requires to perform voting to assess the need for adaption.

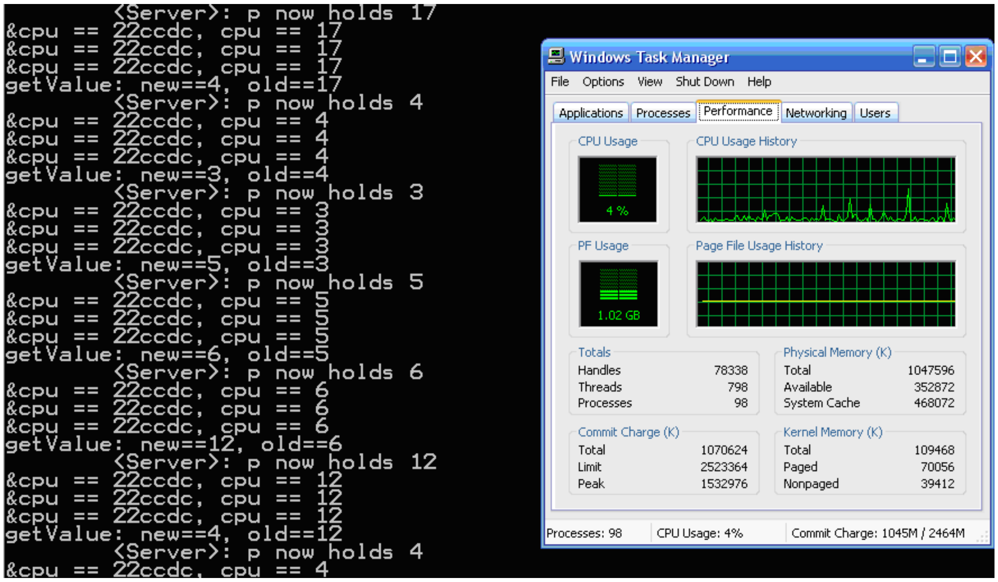


```
#include <stdio.h>
int main(void)
{
        Ref_t int cpu;
        Ref_t int tcpTxRate;

        cpu=0;
        while (1) {
                printf("&cpu == %x, cpu == %d\n",
                        &cpu, cpu);
                if (cpu > 90) break;
                sleep(1);
        }

        tcpTxRate = 70;
}
```

Figure 3. A simple example of the use of RR vars.



```
        <Server>: p now holds 17
&cpu == 22ccdc, cpu == 17
&cpu == 22ccdc, cpu == 17
&cpu == 22ccdc, cpu == 17
getValue: new==4, old==17
        <Server>: p now holds 4
&cpu == 22ccdc, cpu == 4
&cpu == 22ccdc, cpu == 4
&cpu == 22ccdc, cpu == 4
getValue: new==3, old==4
        <Server>: p now holds 3
&cpu == 22ccdc, cpu == 3
&cpu == 22ccdc, cpu == 3
&cpu == 22ccdc, cpu == 3
getValue: new==5, old==3
        <Server>: p now holds 5
&cpu == 22ccdc, cpu == 5
&cpu == 22ccdc, cpu == 5
&cpu == 22ccdc, cpu == 5
getValue: new==6, old==5
        <Server>: p now holds 6
&cpu == 22ccdc, cpu == 6
&cpu == 22ccdc, cpu == 6
&cpu == 22ccdc, cpu == 6
getValue: new==12, old==6
        <Server>: p now holds 12
&cpu == 22ccdc, cpu == 12
&cpu == 22ccdc, cpu == 12
&cpu == 22ccdc, cpu == 12
getValue: new==4, old==12
        <Server>: p now holds 4
&cpu == 22ccdc, cpu == 4
```

Totals	
Handles	78338
Threads	798
Processes	98

Physical Memory (K)	
Total	1047596
Available	352872
System Cache	468072

Commit Charge (K)	
Total	1070624
Limit	2523364
Peak	1532976

Kernel Memory (K)	
Total	109468
Paged	70056
Nonpaged	39412

Processes: 98 CPU Usage: 4% Commit Charge: 1045M / 2464M

Figure 4. An excerpt from the execution of the code in Fig. 3.

The RR var model does not require any special language: Figure 3 is an example in the C language. The portrayed program declares two variables: "cpu", a reflective integer, which reports the current level of usage of the local CPU as an integer number between 0 and 100, and "tcpTxRate", a reflective *and refractive* integer, which reports *and sets* the send rate parameter of the TCP layer. The code periodically queries the CPU usage and, when that reaches a value greater than 90%, it requests to change the TCP send rate. Note that the only non standard C construct is attribute "Ref_t", which specifies that a corresponding declaration is reflective or refractive or both. Through a translation process this code is instrumented so as to include the logics required to interface the cpu and the TCP external devices. Figure 4 shows this simple code in action on our development platform—a Pentium-M laptop running Windows XP and the Cygwin tools.

We observe that through the RR var model the design complexity is partitioned into two well defined and separated components: The code to interface external devices is specified in a separate architectural component, while the functional code is produced in a familiar way, in this case as a C code reading and writing integer variables.

The result is a structured model to express tasks such as cross-layered optimization, adaptive or fault-tolerant computing in an elegant, non intrusive, and cost-effective way. Such model is characterized by strong separation of design concerns, for the functional strategies are not to be specified aside with the layer functions; only instrumentation is required, and this can be done once and for all. This prevents spaghetti-like coding for both the functional and the non-functional aspects, and translates in enhanced maintainability and efficiency.

A special reflective variable is "int _Redundance". The device attached to _Redundance is in this case responsible for providing a trustworthy and timely estimation of the degree of redundancy matching the current disturbances. This device should be in most cases mission-dependent—for instance, for electrical substation automation systems, that could be a sensor able to assess the current electro-magnetic interference. What we consider as a significant property of our approach is that such dependence is not hidden, but isolated in a custom architectural component. In so doing, complexity is partitioned but it is also explicitly available to the system designers for inspection and maintenance. This can facilitate porting *the service* which, as we pointed out in [5], is something different from porting *the code* of a service. Failing to do so can bring to awful consequences, as can be seen in well-known accidents such as the failure of the Ariane 5 flight 501 or, even worse, in the failures of the Therac-25 linear accelerator [16].

In the case at hand—more a proof of concepts than a full fledged product—the device connected to _Redundance is just a task that receives the return values of the majority voting algorithm executed when reading a redundant variable. Such return value is the maximum number of agreeing replicas in the current voting, which is used to compute the risk of failure, i.e., variable r in (1).

In the following section we describe how our tool behaves when memory faults are injected. We shall see that, despite the above naïve design choices, our tool already depicts valuable results.

5 Performance Analysis

In order to analyze the performance of our system, we have developed a simulator, “scrambler”. Our scrambler tool allows to simulate a memory, to protect it with redundant variables, to inject memory faults (bit flips or “bursts” corrupting series of contiguous cells), and to measure the amount of redundancy actually used. Scrambler interprets a simple scripting language consisting of the following commands:

SLEEP s , which suspends the execution of the scrambler for s seconds,

SCRAMBLE n, p , which repeats n times action “scramble a pseudo-random memory cell with probability p”,

BURST n, p, l , which repeats n times action “scramble l contiguous cells with probability p”,

END , which terminates the simulation.

The above commands can be used to compose a complex sequence of fault injections. As an example, the following script, corresponds to the following configuration: no faults for 1 second, then various disturbances occurring with Gaussian distribution[4], then no disturbances again for 5 seconds:

```
SLEEP 1                         // sleep 1 sec
SCRAMBLE 2000, 0.1053992        // scramble 2000 random cells
                                // with probability f(-3)
SCRAMBLE 2000, 0.3678794        // scramble 2000 random cells
                                // with probability f(-2)
BURST 2000, 0.7788008, 10       // execute 2000 bursts of 10
                                // contiguous cells
                                // with probability f(-1)
SCRAMBLE 2000, 1                // scramble 2000 random cells
BURST 2000, 0.7788008, 10       // execute 2000 bursts of 10
                                // contiguous cells
                                // with probability f(1)
SCRAMBLE 2000, 0.3678794        // scramble 2000 random cells
                                // with probability f(2)
SCRAMBLE 2000, 0.1053992        // scramble 2000 random cells
                                // with probability f(3)
SLEEP 5                         // sleep 5 secs
END                             // stop injecting faults
```

[4] The chosen probabilities correspond to Gaussian $f(x) = \exp(-x^2/4)$ for $x = \pm3, \pm2, \pm1$, and 0.

The idea behind these scripts is to be able to represent executions where a program is subjected to environmental conditions that vary with time and range from ideal to heavily disturbed. Scenarios like these ones are common, e.g., in applications servicing primary substation automation systems [8] or spaceborne applications [19].

In the following we describe a few experiments and the results we obtained with scrambler.

All our experiments have been carried out with an array of 20000 redundant 4-byte cells and an allocation stride of 20 (that is, replicas of a same logical cell are spaced by 20 physical cells). In all the reported experiments we run the following script:

```
SLEEP 1
SCRAMBLE 10000, 0.9183156388887342
SCRAMBLE 10000, 0.9183156388887342
SLEEP 3
SCRAMBLE 10000, 0.9183156388887342
SCRAMBLE 10000, 0.9183156388887342
END
```

Concurrently with the execution of this script, 65 million read accesses were performed in round robin across the array. The experiments record the number of scrambled cells and the number of read failures.

Scrambler makes use of standard C function "rand", which depends on an initial seed to generate each pseudo-random sequence. In the reported experiments the same value has been kept for the seed, so as to produce exactly the same sequences in each experiment.

Experiment 1: Fixed, low redundancy. In this first experiment we executed scrambler with fixed (non adaptive) redundancy 3. Table 1 shows the setting of this experiment. The main conclusion we can draw from this run is that a statically redundant data structures provision in this case fails 111 times: In other words, for 111 times it was not possible to find a majority of replicas in agreement, and the system reported a read access failure. The total number of memory accesses is proportional to $3 \times 65000000 \times k$, where $k > 0$ depends on the complexity of the redundant read operation.

Experiment 2: Fixed, higher redundancy. Also experiment 2 has fixed redundancy, this time equal to 5. Table 2 shows the setting of this new experiment. Main conclusion is that the higher redundancy is enough to guarantee data integrity in this case: No read access failures are experienced. The total number of memory accesses is proportional to $5 \times 65000000 \times k$.

Experiment 3: Adaptive redundancy. In this last experiment we enable adaptive redundancy which we initially set to 5. Table 3 shows the resulting setting. Most worth noting is the fact that also in this case no read access failures show up, but the actual amount of redundancy required to reach this result is much lower. Consequently, also the total number of memory accesses, proportional to $(3 \times 64953188 + 5 \times 5631 + 7 \times 26534 + 9 \times 14648) \times k$, is considerably lower. Figure 5 shows how redundancy varies during the first 100000 read cycles. During this time frame no fault injection takes place. This is captured by our adaptation strategy, which decreases redundancy to 3 after 1000 cycles. Figure 6 depicts an interval where several fault injections do take place. These events are detected and trigger several adaptations.

6 Conclusions

As Einstein said, we should "make everything as simple as possible, but not simpler". Likewise, hiding complexity is good, but hiding too much can lead to disasters—history of computing is paved with noteworthy examples. One of the main conclusions of this paper is that software-intensive systems should be built with architectural and/or structuring techniques able to decompose the software complexity, but *without hiding the basic hypotheses and assumptions about their target execution environment and the expected fault- and system models.* A judicious assessment of what should be made transparent and what should stay translucent is necessary. A major lesson learned from the experiences reported here is that one of the aspects where transparency is not an issue is resiliency. In particular, the fault model should be explicit, or severe mistakes can result. Also, as environments do change,

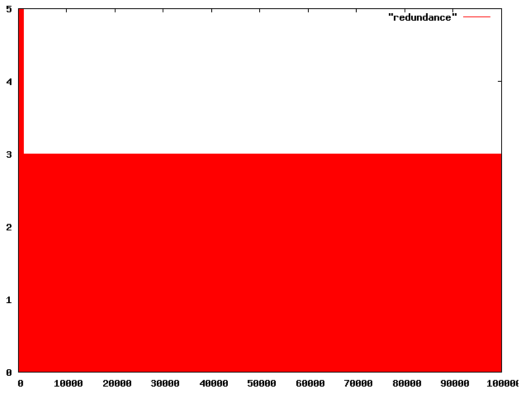


Figure 5. The picture shows how the allocated redundancy varies during the first 100000 cycles of Experiment 3. Note how redundancy drops to its minimum after the first 1000 cycles.

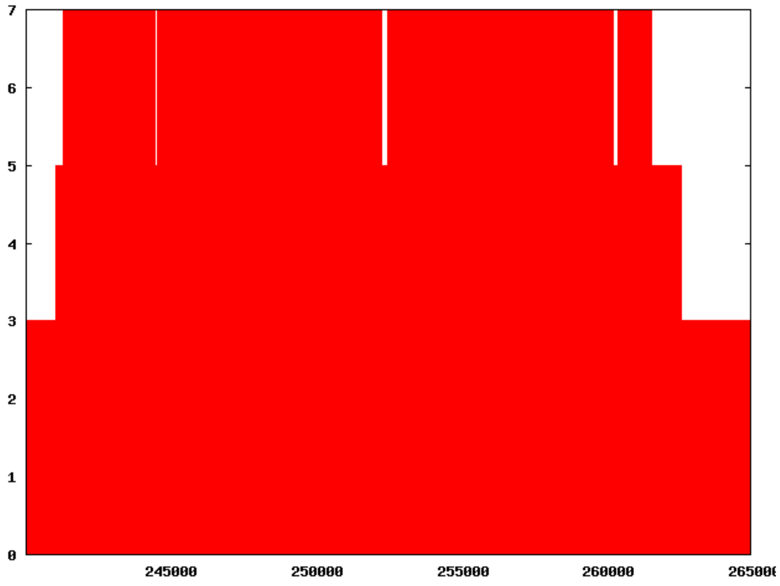


Figure 6. The picture focuses on a section of experiment 3 (namely steps 240150–265000). During such section, fault injection takes place. Accordingly, the allocated redundancy varies so as to match the threat being experienced.

```
$ scrambler faults.in 3 scrub noadaptive
Scrambler::sleep(1)
run 1
run 50001
run 100001
run 150001
Scrambler::scramble(10000,0.918316)
Scrambler::scramble(10000,0.918316)
Scrambler::sleep(3)
run 200001
run 250001
...lines omitted...
run 650001
Scrambler::scramble(10000,0.918316)
Scrambler::scramble(10000,0.918316)
Scrambler::END
run 700001
run 750001
...lines omitted...
run 65000001
36734 scrambled cells, 189 failures, redundance == 3
redundance 3:  65000001 runs
redundance 5:  0 runs
redundance 7:  0 runs
redundance 9:  0 runs
redundance 11:  0 runs
```

Table 1. Experiment 1: Scrambler executes the script in file "faults.in". Parameters set redundancy to 3, select memory scrubbing to repair corrupted data when possible, and keep redundancy fixed.

e.g. because of external events, or because assets dissipate, or because the service is mobile, there is no static allocation of resources that can accommodate for any possible scenario: A highly redundant system will withstand no more faults than those considered at design time, and will allocate a large amount of resources even when no faults are threatening the service.

A core idea in this paper is that software-intensive systems must be structured so as to allow a transparent "tuning" to the constantly changing environmental conditions and technological dependencies. A separated layer must manage monitoring and adaptation, either autonomously or in concert with other system management layers (e.g. the OS or the middleware). Though separated, the intelligence to achieve monitoring or adaptation must not be hidden, but encapsulated in well-defined architectural components. By doing so, it becomes possible to review, verify and maintain that intelligence even when the hypotheses that were originally valid at design time would change.

This paper also introduced and discussed a tool that follows these ideas and provides a simple, well-defined, and effective system structure for the expression of data integrity in the application software. Such tool has a public side, where the functional service is specified by the user in a familiar form–that of common programming languages such as C—and a private side, separated but not hidden, where the adaptation logics is defined. Such private side is the ideal "location" to specify fault model tracking, failure semantics adaptation, resource (re-)optimization, and other important non-functional design goals. An important property to achieve this is reflection, which is useful to define a homogeneous framework where etherogeneous system properties can be monitored and reasoned upon.

We believe that our tool represents one of the practical "ways to enhance the ability of organizations to create processes that are robust yet flexible, to monitor and revise risk models, and to use resources proactively in the face of disruptions or ongoing production and economic pressures" [11, 22], which are sought by the novel discipline of resilience engineering. Software design urgently requires tools, methodologies, and architectures compliant to such

```
$ scrambler faults.in 5 scrub noadaptive
Scrambler::sleep(1)
run 1
run 50001
run 100001
Scrambler::scramble(10000,0.918316)
Scrambler::scramble(10000,0.918316)
Scrambler::sleep(3)
run 150001
...lines omitted...
run 500001
Scrambler::scramble(10000,0.918316)
Scrambler::scramble(10000,0.918316)
Scrambler::END
run 550001
...lines omitted...
run 65000001
36734 scrambled cells, 0 failures, redundance == 5
redundance 3:  0 runs
redundance 5:  65000001 runs
redundance 7:  0 runs
redundance 9:  0 runs
redundance 11:  0 runs
```

Table 2. Experiment 2: Scrambler executes the same script as before; only, redundancy is now set to 5. No failures are observed.

```
$ scrambler faults.in 5 scrub adaptive
run 1
Scrambler::sleep(1)
run 50001
run 100001
run 150001
run 200001
Scrambler::scramble(10000,0.918316)
Scrambler::scramble(10000,0.918316)
Scrambler::sleep(3)
run 250001
...lines omitted...
run 600001
Scrambler::scramble(10000,0.918316)
Scrambler::scramble(10000,0.918316)
Scrambler::END
run 650001
...lines omitted...
run 65000001
36734 scrambled cells, 0 failures, redundance == 3
redundance 3:  64953188 runs
redundance 5:  5631 runs
redundance 7:  26534 runs
redundance 9:  14648 runs
redundance 11:  0 runs
```

Table 3. Experiment 3: Scrambler executes the same script as before; only, redundancy is now adaptive. No failures are observed, but the employed redundancy is mostly of degree 3.

model, which we refer to here as a "new software development."

References

[1] P. Bohannon, R. Rastogi, S. Seshadri, A. Silberschatz, and S. Sudarshan. Detection and recovery techniques for database corruption. *IEEE Transactions on Knowledge and Data Engineering*, 15(5):1120–1136, September/October 2003.

[2] F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *IEEE Trans. on Parallel and Distributed Systems*, 10(6):642–657, June 1999.

[3] V. De Florio. Array associativi, linguaggio C e programmazione CGI. *DEV.*, (27), February 1996. In Italian.

[4] V. De Florio and C. Blondia. Reflective and refractive variables: A model for effective and maintainable adaptive-and-dependable software. In *Proc. of the 33rd EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA 2007)*, Luebeck, Germany, August 2007.

[5] V. De Florio and G. Deconinck. On some key requirements of mobile application software. In *Proc. of the 9th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS)*, Lund, Sweden, April 2002. IEEE Comp. Soc. Press.

[6] V. De Florio, G. Deconinck, and R. Lauwereins. Software tool combining fault masking with user-defined recovery strategies. *IEE Proceedings – Software*, 145(6):203–211, December 1998. Special Issue on Dependable Computing Systems. IEE in association with the British Computer Society.

[7] G. Deconinck, O. Botti, F. Cassinari, V. De Florio, and R. Lauwereins. Stable memory in substation automation: a case study. In *Proc. of the 28th Int. Symposium on Fault-Tolerant Computing (FTCS-28)*, pages 452–457, Munich, Germany, June 1998. IEEE Comp. Soc. Press.

[8] G. Deconinck, V. De Florio, G. Dondossola, and J. Szanto. Integrating recovery strategies into a primary substation automation system. In *Proc. of the International Conference on Dependable Systems and Networks (DSN-2003)*. IEEE Comp. Soc. Press, 2003.

[9] Intelligent content in fp7 3rd itc call. Retrieved on April 24, 2008 from cordis.europa.eu/ist/kct/eventcall3-in-motion.htm. in "Call-3 in Motion: Challenge 4 - Intelligent Content and Semantics".

[10] K. E. Holbert. Single event effects. Retrieved on March 3, 2007 from www.eas.asu.edu/∼holbert/eee460/see.html.

[11] E. Hollnagel, D. D. Woods, and N. G. Leveson. *Resilience engineering: Concepts and precepts.* Aldershot, UK, Ashgate, 2006.

[12] J. J. Horning. ACM Fellow Profile — James Jay (Jim) Horning. *ACM Software Engineering Notes*, 23(4), July 1998.

[13] Institute for Telecommunication Sciences. Telecommunication standard terms. Retrieved on Jan. 31, 2007 from http://www.babylon.com/dictionary/4197/Telecommunication_Standard_Terms_Dictionary.

[14] R. Ladbury. SDRAMs: Can't live without them, but can we live with them? In *Thirteenth Biennial Single Effects Symposium*, Manhattan Beach, CA, April 2002.

[15] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Trans. on Programming Languages and Systems*, 4(3):384–401, July 1982.

[16] N. G. Leveson. *Safeware: Systems Safety and Computers.* Addison-Wesley, 1995.

[17] J. Levine, T. Mason, and D. Brown. *Lex & YACC.* O'Reilly & Associates, 2nd edition, 1992.

[18] M. E. McGrath. *Next Generation Product Development: How to Increase Productivity, Cut Costs, and Reduce Cycle Times.* McGraw-Hill, New York, 2004.

[19] K. K. Oey and S. Teitelbaum. Highly reliable spaceborne memory subsystem. In *3rd Computers in Aerospace Conference*, pages 66–71, San Diego, CA, October 1981. American Institute of Aeronautics and Astronautics.

[20] D. Powell. Preliminary definition of the GUARDS architecture. Technical Report 96277, LAAS-CNRS, January 1997.

[21] C. E. Shannon, A. D. Winer, and N. J. A. Sloane. *Claude Elwood Shannon : Collected Papers.* Amazon, 1993.

[22] L. Simoncini. ReSIST NoE and its WP3 on training and dissemination. In *EWICS-ReSIST Joint Workshop on Teaching Resilient Computing*, Erlangen, Germany, May 2007.

[23] D. J. Taylor, D. E. Morgan, and J. P. Black. Redundancy in data structures: Improving software fault tolerance. *IEEE Trans. on Software Engineering*, 6(6):585–594, November 1980.

[24] P. Van Roy. Self management and the future of software design. *To appear in the Electronic Notes in Theoretical Computer Science (www.elsevier.com/locate/entcs)*, 2006.

[25] T. Van Vleck. Structure marking, 1995. Retrieved on October 8, 2007, from www.multicians.org/thvv/marking.html.

[26] Wikipedia. Latchup. Retrieved on March 3, 2007 from en.wikipedia.org/wiki/Latchup.

[27] Wikipedia. Single-event upset. Retrieved on March 3, 2007 from en.wikipedia.org/wiki/Single_event_upset.

[28] Wikipedia. Soft error. Retrieved on March 3, 2007 from en.wikipedia.org/wiki/Soft_error.