6
Discussion

In this chapter we explore some state-of-the-art research, and compare it to our experiences and results.

6.1.
Recovery strategies

One interesting aspect that is always present in every work is componentization. It is impossible to build a robust, fault-tolerant system without a high level of componentization, as recovery must be done within components [Candea et al, 2003] [Candea and Fox, 2003] [Fox and Patterson, 2002] [Chen at al 2004]. All studied works are heavily focused on internet, mainly web-based systems.

[Candea et al, 2003] proposes a technique based on restarting components whenever they present some kind of malfunction. The idea is that, if a system is designed with independent components, then recovery can be made by simply restarting the damaged component. However, this may not be that simple when a component depends on another one that is damaged – the entire tree of components may need to be restarted. For such a technique to work, we noticed that it is necessary to take care about how the dependence between two components is modeled. If a component C1 depends on component C2, then:

- C1 may not have a direct reference to C2, because if a new instance of C2 needs to replace a defective one, the reference becomes invalid;
- C1 must avoid keeping record of C2 state (for optimization, or other purposes), because if C2 is restarted, such information may become invalid. If this is strictly required, than it is necessary to write code that handles possible change of C2.

If C2 must be unique, it may be implemented as a Singleton [Gamma et al, 1995] which would solve the first issue. However, when C2 must have a

cardinality of one on one to C1, then such pattern cannot be used because a reference to C2 will be required. The Observer pattern [Gamma et al, 1995] can be used so that, whenever C2 needs to be replaced, C1 is warned (i.e., C1 is an observer of C2). Note that this solution mixes functional requirement concerns (represented by C1 operations) and reliability concerns (represented by the observer implementation). To avoid this, a solution to be considered is to use the Proxy pattern [Gamma et al, 1995], so that C1 shall not reference C2 directly, but a proxy, and the proxy may take care of the component replacement by implementing the observer.

Our experiments showed that restarting a component is an effective and a relatively easy way to try to recover from failures only if it is possible to implement the component and a Singleton. In multi-threaded systems, it is necessary to care about concurrency, to guarantee that the restarted component is not in use. However, when there is a web of dependencies between components that cannot be solved by a Singleton (for example, the need of multiple instances), one may consider restarting all the components that depend upon the defective one. This implies that it is better to consider the whole set of affected components as a single one, which will be restarted. Only if this web becomes too wide and complicated, it is better to consider the proxy-observer solution.

[Candea et al, 2003] also proposes the use of monitoring agents, that constantly verify the system overall performance in order to detect damaged components. However, as recovery code can fail too, systems must know how to intelligently retry their recovery. And this is, in our opinion, an important issue: the architecture proposed is too complicated and sophisticated, hard to implement and hard to reuse – as fault detection is usually very tied to the domain of a system. We strongly believe that, to be reliable, recovery code must be simple.

[Candea and Fox, 2003] introduce the concept of crash-only software, defined as “software that crashes safely and recovers quickly”. The claim is that crash-only software can only be developed with a “crash-only design”, which includes the concept of a “crash-only component”. A crash-only component is a component that has a single idempotent power-off switch, and a single idempotent power-on switch4, both external to the component, in such a way that it is

[4] An idempotent switch acts as if used only once, even if used multiple times in sequence

possible to turn-on and turn-off a component as desired. An example of a turn-off switch is the "kill -9" shell command of UNIX. The state of a crash-only component must be kept outside the component, in a “crash-only state store” like the one presented in [Ling et al, 2004]. Some problems arise:

- By introducing crash-only state stores, the problem of information loss and integrity has just been moved one level below – it is no longer a component problem, but it is still a problem. The claim is that internet systems will standardize on a small number of state stores, that will provide such guarantees – i.e., the problem is part of the infra-structure used, and not the programmers concern. However, to what extent can a crash-only store be fully reliable? What if it fails?

- The state store must have ACID properties (Atomicity, Consistency, Isolation, and Durability), because a component’s external power-off switch may be activated anytime, especially while the component is saving its state. However, transactional code usually depends on domain specific issues – it cannot be handled generally. This means that the application code gets more complex, more error-prone and less reliable.

- Coupling code to a specific state store might not be a good design practice. Experience shows that widely adopted frameworks and containers are the ones that smoothly tie themselves to the application, and not the ones that force an application to follow some strict rules, generating a strong dependence application → framework. For example, J2EE web container, with the MVC architecture, has been much more adopted than J2EE EJB container.

However, the concept of crash-only software is very interesting, and very close to what we propose.

[Fox, 2002] proposes something similar to what [Candea et al, 2003] propose about reboots, but in a macroscopic way: he claims that restarting a software from time to time is an efficient way to improve availability. This has already been discussed in the session “ Patterns” of the chapter “ Technologies and tools that support the development of recovery oriented software”. Our experiments showed that, for stateless systems (according to the

definition presented in “ Recovery and Redundancy” section of the “ Recovery Techniques” chapter), restarting is an effective and a relatively easy way to try to recover from failures. The reboots may either be triggered by a failure detection (for example, by a watchdog) of by a time-triggered event – it is not a bad idea to consider restarting a system periodically, even if a failure has not been detected.

[Fox and Patterson, 2002] claim two interesting things:

- It is better for a system to fail often for short time periods than to fail seldom, but then for long time periods. This means that, from the user's (and also from the system under control) point of view, minimizing the MTTR is more preferable than reducing the MTTF as much as possible, as long as the MTTF stays within accepted levels;
- Users tolerate failures to some extent. They accept a degraded quality of service, as long as it does not exceed some acceptable time and does not occur in a frequency higher than is deemed acceptable. The accepted levels vary from software to software, but the paper presents some numbers for e-commerce web-based applications.

These two claims are interesting and presumably correct, but our experiments showed that there is also a 3rd important claim, not mentioned: users do not accept to lose work. This means that a system may crash, as long as it does not cause loss of work. Even if a system fails seldom and for a short period of time, if this failure imposes significant loss of work for the users, they will complain. This concept can be applied to control systems as well – a system being controlled (by software) may admit some software failures as long as they do not impose loss of control or a security hazard and that, when restored, assure that the state of the system is correctly set on the software control system. We used these three ideas for building the systems presented in this work, with very good results regarding to user opinion about the overall quality of the released software.

6.2. Failure detection

Another interesting area of research is dedicated to failure detection. Path-based failure detection is proposed by [Kiciman and Fox, 2004] and explored by [Candea, Kiciman, Zhang, Keyani and Fox, 2003], that presents a J2EE server able to identify failures in the published applications by doing three things:

- Monitoring and logging exceptions thrown;
- Actively visiting web pages (like a crawler) and looking for words like "error", "fault" etc;
- Comparing the J2EE components (Servlets, EJBs, JavaBeans, etc) activation path from a specific web request with valid paths present in an internal database (so that it can be compared in order to identify an "abnormal activation path").

We believe that each approach has several problems:

- An exception-based application throws a lot of exceptions that model expected execution conditions, not only failures. So, it would be necessary to tune the server in order to log only desired exceptions.
- What if a web-page contains the string "error"? Its path would be considered a failure.
- Web requests usually contain specific request parameters that cannot be easily reproduced, and that could cause an unexpected – though valid – path of execution; and where should such a database come from? Who would build it, who would maintain it? Imagine an e-commerce application scenario, like Amazon.com. There are millions of web-pages with a virtual infinite number of possible activation scenarios to visit; and what should be done with applications that are constantly being developed, with the insertion of new components or changes to old ones?

We believe that it is very difficult to separate the "failure detection concern" from the application because it is heavily coupled with the application's domain.

For example, consider an application where a specific ID must be an integer in the interval [0, 10] in such a way that getting 11 as an ID could be considered a failure. This is specific to the application, and can only be checked by adding specific detection code to it. This makes it impossible to generate something like a "non-invasive failure detector". This is partially discussed in [Saha, 2007], when presenting the concept of self-healing software and the differences from fault-tolerant software. Self-healing software must have knowledge about its expected behavior in order to examine whether its actual behavior deviates from its expected behavior in relation to the environment [Saha, 2007]. This is an interesting discussion, because we can argue that self-healing software not only knows what it is doing, but also what it should not be doing – it must be aware of itself, to some extent.

[Andras and Charlton, 2005] presents an interesting discussion about what is defined as "self-aware software". According to them, an adaptive self-aware software system is able to sense itself and maintain itself producing appropriate adaptive behaviors (dependent on the environmental stimuli and on the state of the system). From this point of view, recovery oriented software is a self-aware software, and a self-healing software is recovery oriented. We concluded that redundancy is a key issue to building self-aware software because it enables a "higher level of consciousness" for software. It is possible to use assertions[5], for example, to let software know not only what it is doing, but also what it should not be doing, and also to provide means for it to change the internal or external environments as needed in order to keep providing its expected services. Software with enough redundancy (code and data) can be considered as the first step to building self-healing software.

Our experience showed that a small level of code redundancy (between 9% and 13% of total lines of code in our experiences) and data redundancy (a few columns in few tables, a few extra fields in data structures) can provide a good level of failure detection, and can provide some level of recovery capabilities. Also, the overall impact on the system performance was acceptable for most of

[5] Remember that our assertions do not cancel execution, but trigger some handler to take an action that depends on the kind of assertion that has been violated. They are not implemented using the macro ASSER, or something like it.

the redundancy instruments (less than 10%, when compared to the system compiled without no failure detection instrument turned on), even for the real-time systems. The instruments which impacted significantly in performance were still useful for development purposes, but had to be turned off in production environment.

6.3. N-Version based failure detection

The results presented in this thesis can be compared to other more expensive techniques for developing fail-safe software such as software redundancy (also called N-version development), when more than one development team independently develop different solutions for the same problem [Avizienis, 1995].

Some researches on such techniques suggest that the costs involved are significantly increased, whereas the results in terms of enhancing reliability are questionable [Knight and Leverson, 1986][Scott et al, 1987][Eckhardt et al, 1990][Holloway, 1997][Liburd, 2004].

There are a number of issues that must be considered when separating development teams, like:

- To what extent, and over what phases of the development must the programming teams be isolated in order to increase the independence of the resulting software [Lyu and He, 1993] [Eckhardt et al, 1990]. [Lyu and He, 1993] address this issue by proposing levels of diversity, which may begin from requirements, design and code, or test phases;
- At what point should the versions be integrated into a redundant structure; after each version has been subjected to an independent validation or at an earlier stage to take advantage of multi-version testing as a validation procedure? [Eckhardt et al, 1990]
- A weakness of a software redundant system lies in the decision algorithm. The question of correctness of redundant system depends heavily on the algorithm it uses to determine what output is "correct" given the multitude of outputs by each individual program [Scott et al, 1987][Eckhardt et al, 1990][Liburd, 2004]. In theory, output from

multiple independent versions is more likely to be correct than output from a single version [Liburd, 2004], but this is questionable.

An interesting result presented in [Eckhardt et al, 1990], [Nagy et al, 2006], [Knight and Leverson, 1986], [Holloway, 1997] and [Scott et al, 1987] is the identification of coincident failures at rates much higher than expected. It means that programmers often make similar mistakes. This may indicate that there are complex issues that must be carefully handled, as they have a higher chance of generating faults. These results show that N-version programming cannot solve the reliability problem on its own [Holloway 1997].

Reasoning redundancy seems to be a good way to deal with more complex issues, because it enforces the developers to accurately think about their work using different ways of reasoning, what leads to writing more reliable code. [Eckhardt et al, 1990] reported that few of the independently developed versions passed all test cases used to verify the reliability of a single version. A relevant question is what techniques, procedures or tools were used by each of the 100% compliant versions.

Code and data redundancy are good ways to identify invalid states of execution, which mean invalid outputs [Gotlieb and Botella, 2003]. So, a way to improve the decision algorithm is to enhance the ability for each version to identify problems with its own execution, so that it can inform the algorithm to ignore its output in the final decision. It would be something like saying “do not trust my output; I know the chances it is not correct are very high”.

N-version programming is not completely orthogonal to what is being proposed in our work. In fact, it can be even used together, in order to generate more reliable systems. However, the question about to what extent the extra costs justifies the enhancements in reliability remains.

6.4. Formal methods

We also had very good results with formal methods. We extended the “lightweight formal methods approach” proposed by [Agerholm and Larsen, 1998] and [Fitzgerald and Larsen, 1995] by proposing that formal methods could be used to write the assertions list, what will help the developers to generate

instruments by turning them into executable assertions. We found almost the same results presented in [Hall, 1990], [Fitzgerald and Larsen, 1995] and in [Agerholm and Larsen, 1998], but we could extend their conclusions:

- The use of formal methods did not guarantee that the assertion list was complete and correct. However, it is an excellent instrument for reasoning redundancy, as it leads the developers to "think more accurately using more than one approach" about their work, a practice that is known for generating good results [Hunt and Thomas, 1999]. We believe this redundancy was responsible for the excellent results regarding to the development of code correct by construction.
- It was not necessary for the developers to have a high knowledge of mathematics. Basic logic was enough.
- We did not notice an increase of development costs. In fact, we noticed the opposite, even though it is possible to argue that comparing software developed by different teams using different methods is very difficult.

6.5. Institutionalization aspects

Another interesting issue to discuss is the reaction of the development team to the ideas explored here. As stated before, all teams used virtually the same techniques, tools and procedures during development time. This requires training that was not explicitly measured since all items of the technology described were developed during a period of about six years of experiences. Throughout this time, some people joined the teams, whereas some left the teams. So, it is impossible to even estimate how long would it take for an entire team to be trained in ROS techniques. What we can estimate, now, is how long it takes for a new developer to be trained and fully inserted in en existing ROS development team.

Our experience showed that inexperienced developers react almost the same way when taught about the benefits of developing using ROS techniques. However, experienced developers do not follow a pattern: while some are quickly convinced about the benefits (and also quickly learn), others tend to resist a little longer, or are even impossible to convince.

The training is completely performed using an on-the-job approach and is divided in two parts:

- A week of theory, where the developer is introduced to the tools, practices and procedures. Benefits are also presented, by means of statistical numbers collected from real projects, feedbacks from customers, relevant publications and other developer testimonials. Real projects are used as examples.
- At most two months of on-the-job training, where the developer is trained with real work on real projects (never work on the critical path), but always supervised by a more experienced one. The supervisor does not evaluate, or grade the development of the new collaborator. Her mission is mainly helping and guiding, creating the environment for the development of the new collaborator.

If in two months the developer is not approved, then she is assumed as a lost case, and is cut off the team. Usually, it takes one month for a developer to be considered accurate enough to be completely integrated.