A component-based architecture for self-healing systems

Juho Vuori
Department of Computer Science
University of Helsinki
Helsinki, Finland
Email: juho.vuori@helsinki.fi

***Abstract*— This seminar paper considers some of the problems in creating a architecture for a self-healing system. The presentation is by no means exhaustive. An example of an architecture of a self-healing component is presented and then starting from that architecture, problems of integrating these components are analysed. Completely different problems might arise if different approach was taken.**

I. Introduction

A self-healing system is a system that is able to detect when it is behaving abnormally and then "heal" itself, thus returning to normal behaviour [1]. This is a straight forward definition that allows a lot of interpretations.

Different kinds of self-healing systems have existed for quite a long time. Some RAID storage systems can semi-automatically heal from a hardware failure, only requiring the system administrator to replace a broken harddrive with a functioning one. Watchdog-systems can automatically reboot a computer system that stops producing *heart-beat.* Internet routing system can automatically switch routing packets through alternative links if previously used links stop working. These and most other self-healing properties of existing systems are special purpose. In this paper a more generic approach to self-healing systems is represented. A completely generic self-healing system may not be possible, but it is possible to separate domain-specific and domain-independent parts of the self-healing infrastructure to at least some degree. It is the purpose of this paper to present an architectural solution for a generic, reusable self-healing system.

The paper is structured as follows. Chapter II presents some background concepts and information. Material covered here will be used by later chapters as a basis for the analysis of the presented architecture. Chapter III describes a proposed architecture of a single component in a self-healing system. Chapter IV describes the healing process of a sick component. Chapter V presents a way of building a complete self-healing system from these components. Chapter VI analyses the presented architecture in light of related work and alternative proposals. Finally, chapter VII presents some conclusive remarks.

II. Concepts and background

The discussion of self-healing systems was accelerated by the *autonomous computing initiative* of IBM and the research activity created by it. Self-healing systems problem space is a subarea of generic *autonomous systems* problem space. Other subareas of not always clearly separable from it. Self-healing is very similar to *self-optimisation*. It almost always includes some form of *self-configuration*. It is a good idea to keep the general autonomous systems problem space in mind when studying self-healing systems [2].

Let's consider a very high level distinction to begin with. Computer systems can be divided to *closed* and *open systems*. Behaviour of closed systems is, at least in theory, completely known by its designers. Typically they are designed and implemented by a single or small number of development groups. Open systems, in contrast, are not completely understood. They contain components that are possibly designed by unrelated groups. Designers of a component have no way of ensuring that other components behave the way they would want them to. In some cases open systems might even contain components that deliberately misbehave in order to crash other components or cause other unwanted consequences. In the main part of the paper, only closed systems are considered.

Next, more fine grained step is to analyse computer systems. *A system* consists of *components* that are connected to eachother in some specific way. This *configuration* of connections is called the *architecture* of the system. Each of these components are in turn built of *sub-components*. Internal structure of a component is referred to as the architecture of the component. In this paper, these two meanings of the word architecture are distinguished by referring to the first meaning with the term *system architecture* and the second meaning with the term *component architecture*.

A self-healing system can be seen as a system consisting of two general functions: *computation* and *coordination*. Computation is the primary function of the system. Coordination is the process of arranging different computational units to do their tasks in a way that makes the system as a whole function as designed. Coordination is the part of a system where its healing and other autonomy properties are implemented. How well the coordination can be automatised, defines how autonomic and especially, how self-healing a system is. For example, a self-healing distributed database system might consist of a bunch of replicas each accepting SQL-queries, analysing them, and returning the corresponding data. Clients using this system can connect any of the replicas they wish. These clients and servers of the system are all

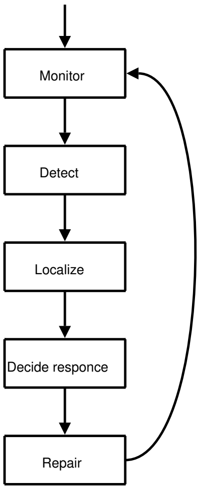

Fig. 1. A monitoring process executes in a loop. It first monitors the computation. After detecting a fault, it localises it, decides what to do, repairs the fault and returns back to the monitoring stage.

computational units that implement the primary function of the system. In order for such a system to work well, some form of coordination should exist to arrange the clients to query the least loaded servers (*self-optimisation*) and to arange the clients not to query broken servers at all. If broken servers are detected, they should be repaired, if possible (self-healing). In cases where the self-healing fails, the system administrator should be notified somehow as a final resort. A system that is able to ask system administrator to help, when needed, is called *an assisted-healing system* [1].

The coordination is traditionally performed by a human system administrator in non-autonomic systems. In autonomic systems, the coordination can be performed by a dedicated piece of hardware, a separate software process or it can be embedded into the application code. The best results are probably achieved by a combination of all three approaches. Conceptually, coordination of an autonomic system is best understood as a separate process called *the monitoring process*. The stages of execution of the monitoring process are illustrated in figure 1. The process will be further analysed below.

The self-healing functionality of a system can be implemented *externally* or *internally* to the application. In a purely external self-healing system, the application is not aware of the monitoring process. In an internal self-healing system, the monitoring process is completely embedded into the application.

The external approach to self-healing is often called *architecture-based self-healing*. It is the most reusable way of implementing self-healing. It might also sometimes be the only possibility, if the self-healing system contains legacy code, which cannot be modified. The problem with external self-healing architecture is that monitoring and repairing possibilities are limited. Monitoring is often limited to simply checking that replies to service requests are returned in time. Repairing is often limited to simply restarting the faulty component. Internal approach allows a more fine-grained control of both monitoring and repairing processes, but such an implementation is not as easily reusable.

Internal self-healing is, in contrast, not easy to implement in a reusable way. The advantage is that the monitoring process can be given, and can dynamically acquire, intimate knowledge about the system. This allows more sophisticated fault detection, localisation, and reparation.

A combination of internal and external self-healing is a tempting approach. There are two ways to implement the combination. A system might run an external monitoring process, but the components are required to cooperate in monitoring and repairing with a fine-grained and standardised self-healing interface. The other combination approach uses embedded monitoring processes. In this approach, each component includes a monitoring process, and all of these monitoring processes provide an interface that other monitoring processes can use to communicate their state. The monitoring process is primarily interested in monitoring the component itself, but in case of trouble it can request neighbouring components to help in repairing a problem. The latter approach is the one used in the architecture presented in this paper.

III. Architecture of a self-healing component

Propositions for architectures of self-healing systems vary a lot. It is not possible to cover all the material in this paper. A *connector-based architecture* for elements of a self-healing systems is presented here [3] [4]. The architecture is developed by a research group in the Texas Tech University and should be taken as a example of different approaches and as a basis for analysis of other architectures. Connector-based architecture is primarily an architecture of a single component of a self-healing system. However, it extends to complete systems in a straightforward way.

A component is structured as active and passive *objects* and *connectors* between them. Active objects are also called *tasks*. Passive objects could be files and other entities with no thread of control of their own. It is safe to simplify the picture a bit and just view the system as a collection of tasks connected to each other through objects called connectors. Any communication to elements outside the component is also performed through connectors. This is illustrated in figure 2.

This is the computational part of a self-healing component. It is referred to as *the service layer* of a self-healing component. To complete the architecture, a special *healing layer* is added on top of the service layer as illustrated in figure 3.

The healing layer *monitors* the service layer and *adjusts* it in various ways if faulty states are detected. The healing layer may also provide services to other components in order to coordinate the healing process, but it does not participate in providing the primary services of the component. An overview of the architecture is shown in figure 3.

The self-healing functionality of a system is build around its connectors. There are several types of connectors, but all share the same basic functionality, they pass messages between other objects. The idea is that whenever a connector receives

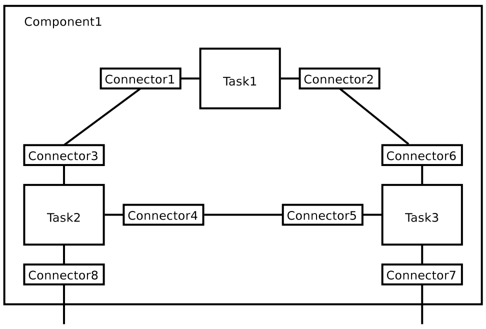

Fig. 2. Task objects connected to each other with connector objects. Connectors 7 and 8 connect to elements outside this component such as devices or other software components.

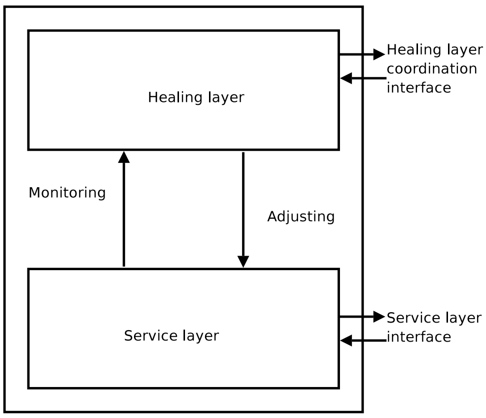

Fig. 3. Rough picture of the architecture of a self-healing component.

or sends a message to or from another connector or an object, it notifies the healing layer of the component. The healing layer maintains a model of the component behaviour and compares it to the observed message passing patterns. If an object responds in an unexpected way to a received message, then the healing layer knows something is going wrong and considers actions to take.

The healing layer is organised as following components: *component reconfiguration plan generator*, *component repair plan generator*, *component self-healing controller*, *component monitor*, *component reconfiguration executor* and *component repair executor*. The rough internal structure of the healing-layer is illustrated in figure 4.

IV. Healing layer and the self-healing process

The self-healing mechanism is based on five steps: monitoring, detecting, localisation, response decision and reparation of the problem. In the presented architecture, monitoring is done with full cooperation of the service layer; the connectors just pass information about the messages passed through them to the component monitor. The component monitor contains state charts of the behaviour of tasks and if the state charts and the observed message passing patterns don't match, a faulty state is detected. Localisation is also easy in the presented architecture, as the problem is always assumed to be in the object that responds in an unexpected way.

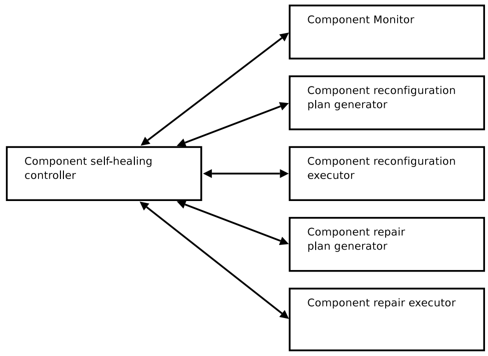

Fig. 4. The structure of the healing layer.

Once a fault is detected and localised, the component monitor then notifies component self-healing controller, which begins the response decision and repairing phase. Repairing a sick object might have side effects if it kept on communicating with other objects during the repairing process. The first step, therefore, is to isolate the sick object from healthy ones. Then the sick object is healed and finally the healed object connected to the rest of the system again.

Isolation is performed by reconfiguring all the objects connected to the sick object. Ideally there is a redundant object in the system and the connected objects can just start using services of that instead of the sick object. However, some objects are vital to the component and may not be replaceable. In this case the component stops functioning during the healing phase and the neighbouring components are notified. In the most unlucky case, the whole system might be useless during the healing of a sick object. Also, all transactions involving the sick object must be rolled back. Component reconfiguration plan generator maintains information about this. It knows which objects inside the component to notify and also which neighboring components to notify when an object is being repaired. Component reconfiguration executor is responsible of sending the notifications.

Next step is to repair the object. As with reconfiguration, the component repair plan generator maintains information about how to repair each object in each situation and component repair plan executor executes the plan. The reparation process differs from object to object, but some examples are:

- reinitialisation of the sick object;
- replacing the object with functionally identical alternative implementation;
- virus-scanning;
- running a file system checker or other data structure regeneration process.

Final part of the reparation of the sick object is to test the repaired object with some test data, which is part of the repair plan.

Now the object is repaired, but the component still needs to be reconfigured to the original state. This phase is performed by component reconfiguration plan generator and component reconfiguration executor in a similar way as the original

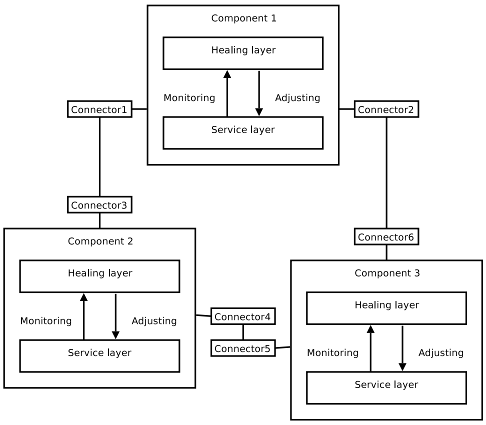

Fig. 5. The structure of the whole self-healing system.

isolation of the sick object. If everything went fine, the system is now in its initial state and works properly. In the presented architecture, the last two parts of the self-healing process, response decision and reparation are done in three distinct phases: reconfiguration before repair, repair and reconfiguration after repair.

If, for some reason, the reparation process did not succeed, the only thing left to do is to inform the system administrator so that manual intervention can be done.

V. Self-healing system architecture based on self-healing components

Above, a component in a system was structured to tasks and connectors between them. Similarly a whole system can be structured to components and connectors between them. A system built of self-healing components and their connectors is illustrated in figure 5.

As explained above the self-healing of a system happens almost completely inside its components. No extra self-healing components need to be added. The self-healing process is a distributed process that emerges when the healing layers of all the components of the system cooperate with each other.

In detail, when a component in a system fails, the component self-healing controller notices it. It then prepares for reparation by isolating the faulty object and notifying neighbouring components of its new degraded state. It is the responsibility of the component self-healing controller of the neighbouring components to then reconfigure themselves to adapt to their neighbours degradation and if necessary inform their neighbours and so on. The reparation is performed alone by the sick component and returning to normal system state after the reparation is done similarly to entering the degraded state. The now-healed component informs its neighbours that it is now ready to serve them normally. The neighbours reconfigure themselves and notify their neighbours and finally the system is healed.

VI. Analysis, alternative proposals, open questions and related work

The presented architecture for a self-healing system is just one proposal. There are some problems with it that other architectures might solve better. In this chapter, some of the strengths and weeknesses of this architecture are analysed.

A. *Component-based versus system-wide healing*

The discussion in this paper is based on component architecture. Only secondarily is the system architecture considered. The architecture proposed here is essentially a *bottom-up* self-healing architecture. The problems are monitored at as low level as possible. A completely different, *top-down* approach is to monitor the system as a whole. This way faults would be discovered in the general behaviour of the system and only afterwards they would be localised to faulty behaviour of a single component or an object. The advantage of the presented architecture is that localisation of faults is very simple, but the disadvantage is that a whole system can have emergent behavioral patterns that are faulty even though every component on its own behaves correctly. Such problems can arise for example if there are design flaws in the architecture of the system. Especially as the presented architecture is dynamic and may change during (possibly multiple simultaneous) failure reparations, verifying the correctness of the architecture is a demanding task. More top-down minded propsals of self-healing architectures are being researched extensively [8] [9] [10].

It is also reasonable to ask, whether the failure localisation proposed by the presented architecture is too simplistic. Is it always so that the failure is located in the object whose message-passing pattern first mismatches with the model of the component self-healing monitor. The anomalous message might, for example, be caused by a corrupted data structure, and the actual problem could then be located in the object that first corrupted the datastructure. An ideal self-healing component would have noticed this while corrupting the datastructure, but it is probably unrealistic to try to design a healin layer that would detect all such failures.

The presented architecture is based on an idea that the interfaces between objects are the best place to try to detect misbehaviour of the components. The referenced papers do not present arguments in favor of this approach. One can, however, think of at least three reasons why connectors are a good place to monitor system behaviour.

- System interconnections inside an object are often complex, hence it is often difficult to distinguish between faulty and non-faulty states of an object. However its interface with other objects is often designed to be simpler so fault detection should be easier there.
- Even though an object could detect that it is in a faulty state, it is often difficult to recover. It might be necessary to restart the object anyway, and this can as well be done from outside the object.
- A component might be a third party product — essentially a black box — behaviour of which is difficult to analyse

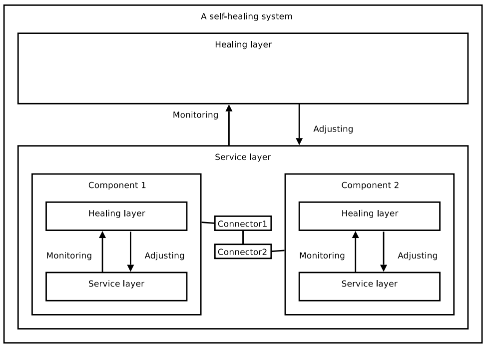

Fig. 6. Generalisation of the presented architecture to system level. This generalisation is not proposed by any of the referenced papers.

or control. The output of such component is observable anyway so it is the first location where faults of such a component is detectable.

Another issue is what happens when a complete component, including its self-healing layer, fails. The architecture doesn't propose any way to heal from such a situation. It is imaginable that the proposed architecture of a self-healing component could be generalised to system level, so that the whole system would consist of a system-wide service and healing layers as illustrated in figure 6. The system level healing-layer would then be responsible in monitoring the system as a whole and restarting or otherwise repairing completely faulty components. However, whether this or some other architecture would be best for system-level self-healing is not covered in the referenced papers.

B. Application domains of self-healing systems

A self-healing elevator coordination system has been designed using the presented architecture [4]. Otherwise than that, it is not studied in the referenced papers, what is the ideal application domain of the architecture. Application domain of a self-healing system architecture can be classified based on its *fault model* and its *design context* [7].

No system can heal from all imaginable failures. Faults can be classified based on their *duration*, *manifestation*, *source*, *granularity* and their *profile expectations* [7]. It is different to build a system that is able to heal from *permanent faults* in hardware than to build a system that is able to heal from *transient faults* such as race conditions. Permanent hardware faults can cause whole components to fail completely, and will require system-wide healing mechanisms. Transient faults may be easily handled locally by just retrying a failed operation. A fault may cause a system to crash immediately or it may cause datastructure corruptions that can be observed from external system behaviour only long after the initial fault occured. These different types of fault manifestation require different detection and reparation mechanisms.

The design context is classified to following subareas: *component homogeneity*, *behavioral predetermination*, *user involvement*, *system linearity* and *system scope* [7]. Component homogeneity will be further analysed in next section. Behaviorally predetermined systems, are ones whose behaviour can be exactly modelled in advance. Self-healing in such systems is easier to implement than in systems whose behaviour is not predetermined. In the latter case, different approximations must be used and the systems runs into a risk of false detections of failures. By System scope, large distributed systems are distincted from single-node computing systems. The proposed architecture fits nicely into distributed systems, where complete nodes can be replaced by others during the healing phase. It possibly can be used in single-node systems as well, but the healing process becomes more limited as there are less redundant components available.

C. Legacy and open systems

The proposed architecture is primarily designed for closed systems. It is applicable to open systems as well, but additional problems arise with them [6]. Self-healing is a central part of designing open systems, as the designers cannot trust that other components of the system function as claimed. Indeed, Internet routing system, for example, implements some forms of self-healing to route packets around broken links.

Components of an open system are often non-homogenous. This means that there exist different implementations of a component, which, however, slightly differ in their behaviour. This often complicates the self-healing process, as it may not be straight forward to use other components as spares for a broken component [7]. The advantage of having different implementations of a component is that a bug in an implementation probably does not exist in another implementation. Therefore if a component fails to do certain computation due to a software bug, it is likely that another component performs well in the same situation. Building self-healing open systems is largely a standardisation issue [2]. Other significant problem with open systems is security.

It is sometimes necessary to include components in a system, source code of which is not available and which are not inherently built as self-healing components. This kind of legacy software introduces similar problems as open systems. There is work being done on injecting self-healing code to legacy software that is not designed to be self-healing [5]. The scheme isn't specifically designed with the proposed architecture in mind. The referenced paper explicitly handles java software only. It is not clear whether something similar could be implemented with software written in C-like languages.

Luckily, the presented architecture can be applied to legacy components quite well without any code injection. What needs to be done, is to implement new connector objects around the legacy component. These connectors can be monitored the same way as other connectors. Internal message passing of the component is not monitorable in this way, of course.

D. Reusability

One potential critisism is reusability of the presented approach. Self-healing functionality is embedded into the components of the system. Does the self-healing functionality need to be rewritten each time a new component is developed? The question is not analysed in the referenced papers, but it seems that the component self-healing controller, the component reconfiguration executor and the component repair executor can be buit as generic components. The component-specific objects in the healing layer are the component monitor, the component reconfiguration plan generator and the component repair plan generator. It seems that it is even possible to split the component monitor to a generic monitoring object and a component-specific object that maintains information about which monitored sequences indicate failure. This requires generic and flexible standards on reconfiguration plan and repair plan languages.

One problem is that even with the above presented splitting of a system to generic and component-specific parts, the component specific parts may end up being very large. It is proposed [4], that the component monitor holds a simple state chart of faulty and non-faulty communication sequences of the objects. However, one might reasonably ask whether this is realistic at all. If it is, the state charts of complex real-world systems will be either huge or very limited in their capability to distinguish faulty from non-faulty behaviour. There are also proposals of using machine-learning or other intelligent algorithms for deciding whether a given behaviour of a system is faulty or non-faulty. It is not clear how well these more sophisticated algorithms fit into the proposed architecture.

VII. Conclusion

There are two main questions that self-healing systems architecure research should answer. The first one is: where in a given software system its self-healing functionality should be located. Two main opposing schools of thought are:

- The self-healing functionality should be located on top of the whole system independent of its structural components;
- The self-healing functionality should be embedded into its components.

There is a clear trade-off between these points of view. If as in the first proposition, the self-healing functionality is on top of the whole system, it is easily reusable. However, system failures are not generic, but they are spesific to the behaviour of a system. A behaviour that would be considerd faulty for a component is acceptable for another component. Therefore, if the self-healing functionality is embedded at lower levels, it can have more fine-grained understanding of faulty and non-faulty behaviour of systems. The proposition presented is primarily a low-level approach, but tries to bridge the gap between the two approaches.

The other question is independent of the first one: How to build a self-healing system. Self-healing systems are complicated. The detection of failures, deciding how to react on them may involve complex intelligent algorithms. Repairing the system automatically while it is running involves careful modelling of the system and so on. Given all the necessary algorithms to perform these tasks, it is still a challenge to wire these together to form a whole self-healing system.

The answers given in this paper are only examples of current research. The author makes no claim that the presented architecture would be any better or worse than other propositions.

References

[1] D. Ghosh, R. Sharman, H. R. Rao, and S. Upadhyaya, "Self-healing systems - survey and synthesis," *Decis. Support Syst.*, vol. 42, no. 4, pp. 2164–2185, 2007.

[2] J. O. Kephart, "Research challenges of autonomic computing," in *ICSE '05: Proceedings of the 27th international conference on Software engineering*, 2005, pp. 15–22.

[3] M. E. Shin and D. Cooke, "Connector-based self-healing mechanism for components of a reliable system," in *DEAS '05: Proceedings of the 2005 workshop on Design and evolution of autonomic application software*. New York, NY, USA: ACM Press, 2005, pp. 1–7.

[4] M. E. Shin, "Self-healing components in robust software architecture for concurrent and distributed systems," *Sci. Comput. Program.*, vol. 57, no. 1, pp. 27–44, 2005.

[5] M. M. Fuad, D. Deb, and M. J. Oudshoorn, "Adding self-healing capabilities into legacy object oriented application," in *ICAS '06: Proceedings of the International Conference on Autonomic and Autonomous Systems*. Washington, DC, USA: IEEE Computer Society, 2006, p. 51.

[6] S. Fickas and R. J. Hall, "Self-healing open systems," in *WOSS '02: Proceedings of the first workshop on Self-healing systems*. New York, NY, USA: ACM Press, 2002, pp. 99–101.

[7] P. Koopman, "Elements of the self-healing system problem space," 2003. [Online]. Available: citeseer.ist.psu.edu/koopman03elements.html

[8] N. Combs and J. Vagle, "Adaptive mirroring of system of systems architectures," in *WOSS '02: Proceedings of the first workshop on Self-healing systems*. New York, NY, USA: ACM Press, 2002, pp. 96–98.

[9] R. de Lemos and J. L. Fiadeiro, "An architectural support for self-adaptive software for treating faults," in *WOSS '02: Proceedings of the first workshop on Self-healing systems*. New York, NY, USA: ACM Press, 2002, pp. 39–42.

[10] D. Garlan and B. Schmerl, "Model-based adaptation for self-healing systems," in *WOSS '02: Proceedings of the first workshop on Self-healing systems*. New York, NY, USA: ACM Press, 2002, pp. 27–32.