Self–Healing Protocol Implementations

Extended Abstract
Christian Tschudin and Lidia Yamamoto, Oct 2004

In this talk we consider an extended failure model for communication software: Instead of having to handle only external failures like lost or corrupted data packets, invalid routes and crashing nodes, we request the software to also handle internal errors. Internal errors can be the unfaithful execution of some instructions or the partial loss of the code base. We then ask whether software is able to detect such internal errors (self-monitoring), is able to continue operation despite such errors (resilience) and is able to correct such errors (self-healing). As a first step towards a better understanding of this problem, we restrict our attention to a simple "knock-out" criteria: The challenge is to demonstrate a program that continues correct execution despite removal of an arbitrary instructions. Having such resilient protocol implementations would permit to distribute the corresponding execution circuits over a network without having to worry about unreliable execution platforms in the same way as we expect current protocols to handle unreliable communication channels.

Reasons to "harden" Communication Software

Resilience and self-healing ability are essential properties of a truly self-organizing network, where functional and coherent protocol structures must emerge out of basic protocol submodules. A resilient network must be able to detect and replace misbehaving software at run time, while continuing to provide the service, although perhaps less efficiently during the transitory repair phase.

In most commonly encountered current computer systems there is always a risk of service disruption due to buggy or malicious code. The underlying software systems are usually not robust to misbehaving code, and are unable to autonomously resume their normal behavior after such misbehaving code has been installed. This fragility stems from the implicit assumption that all code should be well-behaving, predictable and correct. This assumption is unrealistic, as it can be observed daily in the form of disruptive software bugs, viruses, worms, and attacks of various kinds. But also at the hardware level we envisage that errors can occur.

If true service resilience can be achieved, it means that altering a single instruction will not do any harm to the protocol in question. A consequence of this is that it becomes in theory possible to disperse the code of this resilient protocol, such that each atomic instruction would be carried out by a different processor. Register values could be shipped in packets between the nodes. Since the protocol is robust against the loss of a single instruction, it is now robust against the crash of any of the processors involved. In practice such partitioning would not occur on a single instruction basis but at the level of modules or code compartments. In this case we would like that any compartment crash be tolerable, while keeping instruction-level robustness inside the compartment. Robustness can be examined at different levels.

The applications of resilient and self-healing protocols are numerous: they would enable safe automated installation of new protocols, protocol upgrade, run-time customization of protocols to adapt to different network situations, distributed protocol implementations in sensor networks, spray computers, support for ambient intelligence and other networks of small devices, the dynamic placing of middle-box elements such as proxies and caches, and so on.

Surviving a Code Knock–Out Attack

Attempting to solve the problem by installing supervision capabilities into a network leads to a circular problem: The supervision, which is also implemented in software, is subject to the same problems. In its simplest form this circular problem can be reduced to a program that continues execution despite knocking out an arbitrary instruction.

Conventional sequential programming styles are not suited for surviving such a "knock out" attack. Most instructions on the main execution path, regardless whether we look at them at the level of assembler or a high level programming language, are single points of failure: Removing or replacing them will in most cases disrupt the whole service they are implementing. In order to

Dagstuhl Seminar Proceedings 04411
Service Management and Self-Organization in IP-based Networks
http://drops.dagstuhl.de/opus/volltexte/2005/98

avoid single failure points we must turn to execution environments where multiple execution paths exist. Like in the case of basically unreliable transmission of messages where messages can be lost, reordered etc., we assume that some subset of the execution paths of the protocol software are executed in an unreliable way. We then ask whether communication software can be written such that it is able to recover itself in such circumstances.

As a first approach to the problem we take inspiration from metabolic pathways in cells. These chemical processes are highly interlocked and surprisingly robust. This is of major interest to the pharmaceutical industry that is faced with the problem of identifying the multiple change points in a metabolic pathway in order to alter a cell's production levels (e.g. reproduction of a virus, cancer cell, etc.), where a single inhibition point is in general hard to find.

A Resilient "Message Duplicator" Program

In the talk we will present a simple program in a communication context whose task is to duplicate each incoming message i.e., to double the message frequency. This program is expressed in the Fraglet model [2]: Fraglets are computation fragments i.e., small rules inside a multiset that either interact with each other or which are subject to some independent transformation. A simple prefix language is used to specify a fraglet's behavior. Fraglets have a natural representation as packets where the packet headers correspond to the fraglets' prefixes. Nodes communicate with each other by the exchange of fraglets.

The message duplicator program can be made resilient by duplicating most of the fraglet rules. In normal operation, each instance of a rule will receive half of the processing load. If one rule is knocked out, the other will take the whole load. In order to add self-monitoring capabilities, we enhance each rule with an additional side effect of leaving a trace of its execution. This means that for every successful execution of a fraglet rule, another fraglet will be created. By careful placement of such tracing rules and by adding fraglets that merge this stream of tracing fraglets, we are able to tell which rule was knocked out. More specifically, this diagnosis capability also applies to the tracing rules for which we have to apply the knock-out criteria, too. Overall, our program is capable of providing continuous service *and* to signal which fraglet of its code base was removed, in a resilient way. See [1] for a detailed account as well as a communication protocol example. Still pending research is the challenge to also add self-healing capabilities where the signal is used to re-install the knocked-out fraglet.

Related Work and Outlook

Several areas in computer science have looked into the robustness of software systems but not at the instruction level. Fault tolerance is one such area where the related approaches (replication, state persistence etc) rely on the faithful execution of the core logic. Self-testing and self-correcting software is another area, which addresses malfunctioning software, but does not address its own malfunctioning either. Promising techniques can potentially be found in the field of quantum computing where methods have been devised to let a (fixed) quantum circuit compute reliable results despite partial errors in its execution. Currently it remains an open issue whether classical protocol implementations can be robustified like in the example give above, or whether a special encoding of protocol execution paths is necessary like in quantum computing.

References

[1] C. Tschudin and L. Yamamoto. A Metabolic Approach to Protocol Resilience. In *Proceedings of 1st Workshop on Autonomic Communication (to appear, LNCS 3457)*, Berlin, Germany, Oct 2004.

[2] C. Tschudin. Fraglets - a Metabolistic Execution Model for Communication Protocols. In *Proceeding of 2nd Annual Symposium on Autonomous Intelligent Networks and Systems (AINS)*, Menlo Park, USA, Jul 2003.