A Metabolic Approach to Protocol Resilience

Christian Tschudin[1] and Lidia Yamamoto[2]

[1] Computer Science Department, University of Basel
Bernoullistrasse 16, CH-4056 Basel, Switzerland
`Christian.Tschudin@unibas.ch`
[2] Hitachi Europe, Sophia Antipolis Laboratory
1503 Route des Dolines, F-06650 Valbonne, France
`Lidia.Yamamoto@hitachi-eu.com`

Abstract. The goal of this research is to create robust execution circuits for communication software which can distribute over a network and which continues to provide its service despite parts of the implementation being knocked out. Like packets that can be lost (which can be recovered by the appropriate protocols) we envisage an environment where parts of a protocol's execution can be lost. The remaining implementation elements should continue to operate and be able to recover by themselves for restoring full services again. Based on a chemical execution model, we show a few initial examples of packet processing functions that are robust against the knock-out of any single instruction. These examples illustrate how the model can be applied to implement resilient communication protocols, to which we add regulatory signals that can be used to steer the protocols' code basis.

Keywords: resilient communication software, autonomic communication, bio-inspired networking, active networking, Fraglets.

1 Introduction

Autonomic Communication [1] is a long-term research initiative aimed at the study of the self-organization of network elements, toward their autonomous behavior and automated evolvability. Autonomic networks must be self-managing, which includes self-monitoring and self-healing, among other self-* properties. Several areas are concerned, including security, trust, stability, resilience, control, programmability, behavior composition, and context awareness.

Two important and complementary goals of autonomic communication are resilience and self-healing capacity: resilience against internal failures and misbehavior, and self-healing ability to recover from such abnormal conditions.

Resilience and self-healing ability are essential properties of a self-organizing network, where functional and coherent protocol structures must emerge out of basic protocol submodules. The system 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 full service disruption due to buggy or malicious code. The underlying

In:
Proceedings of the First Workshop on Autonomic Communication (WAC 2004)
Berlin, Germany, October 2004. M Smirnov (Ed.),
Lecture Notes in Computer Science (LNCS 3457), pp. 191-206.

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.

The current methodology for the design of protocols is not better. It relies on the strong assumption of full reliability of the executing components. The fragility of this approach is easy to demonstrate: remove the processor, remove the software module, remove a procedure, or even a single instruction. All these will with high probability lead to the protocol failing to provide the intended communication service.

We concentrate on the last aspect, i.e. the impact of removing a single instruction, or small code fragment, and formulate a first goal: a protocol implementation should be robust enough such that perturbing any single instruction does not lead to wrong protocol behavior. Note that the definition of a single instruction or code fragment depends on the instruction set used, and will be explained in Section 4.

The second goal is to be able to detect and correct wrong protocol behavior, provided that the amount of error is below a threshold. Such self-healing ability is essential to maintain the first goal (resilience) during a potentially long period of operation. Otherwise errors could accumulate, leading sooner or later to service disruption.

This is analogous to the desired properties of forward error correcting codes where a single bit flip still permits to recover and correct a message, or a reliable transfer protocol where message retransmission can be requested, which in our case would correspond to rectifying an incomplete protocol software.

In other words: Like the 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.

If this 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 [2], 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.

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.

We build upon previous work on fraglets [3] as it permits to easily demonstrate a simple example of a robust piece of software. The fraglet model comprises a unified code/data format and execution engine inspired by metabolic networks in molecular biology.

The rest of this paper is organized as follows: Section 2 discusses related techniques including self-testing software, fault tolerant systems, resilience in today's protocols, and so on. Section 3 describes parallel execution frameworks for communication software and introduces the fraglet model and instruction set. Section 4 gives a few initial hints on how resilient protocols can be constructed such that the loss of a (fraglet) rule has no impact on the service but leaves some traces behind which can be used to trigger a self-healing process. Section 5 presents our current conclusions and ideas to motivate a new branch of autonomic communication dedicated to resilient and self-healing code for autonomic network protocols.

2 Related work

2.1 Protocol robustness today

Resilience is a key requirement for any network. At the hardware level it is common to have redundant links, and redundant parts in core-network routers. At the software level, most current protocols and network services incorporate some form of robustness. We give a few examples below.

Robustness to link or node failure is generally achieved by rerouting traffic to alternative paths. In OSPF (Open Shortest Path First), node or link failures are detected via link advertisement messages, and new routes are recomputed accordingly. In BGP (Border Gateway Protocol), as well as in MPLS (Multi-Protocol Label Switching) route restoration is achieved via backup paths, such that service can be preserved during failure of the main path.

At the transport layer, TCP recovers from packet loss, duplicate packets, and congested paths, via an integrated retransmission and congestion control mechanism. Other adaptive transmission schemes are able to recover from similar disturbances by using mechanisms adapted to the nature of the traffic they transport, such as real-time streaming, or loss-tolerant voice/video.

The DNS (Domain Name Server) resolution service is a key pillar of the Internet. Resilience is a paramount concern, and is achieved through redundant servers.

Resilient Overlay Networks (RON) [4] are application-layer overlays on top of the current Internet. They seek to improve end-to-end reliability and performance by dynamically avoiding overloaded or faulty paths. RON nodes monitor path quality in order to detect and select higher quality paths. This makes it possible for the end systems to self-organize into more robust topologies that could not otherwise be offered by the standard Internet routing mechanisms.

These classic robustness protocols focus on external events such as node or link failure, link errors or congestion and are not robust to failure of the protocol implementation itself, except if the failure of an implementation on a node corresponds to the failing of the full node.

2.2 Fault tolerance in distributed systems

The complexity of distributed systems and their dependence upon the underlying hardware and network infrastructure expose them to several possible faults. Fault tolerance must therefore be an inherent part of their design, so that they can keep delivering the intended services with acceptable levels of performance and safety, even in the presence of failures.

There are multiple techniques for fault tolerance in distributed systems [5–8], but most of them use variants or combinations of two main building blocks: state persistence and redundancy.

State persistence can be achieved via checkpointing and/or logging. A checkpoint is a snapshot of the process state at a given execution point. A log contains incremental state changes. Both can be used to resume execution after a failure, at the latest stage for which state has been consistently saved. This can diminish the impact of the failure.

Redundancy can take the form of multiple copies of the same process running on different machines, or multiple versions of a software component which implement the same functionality in different ways. For example, if failure of a version is detected, one can substitute the bad version with another one and try again.

A typical way to implement a fault-tolerant service is by replicating the servers at several independent locations and coordinating the updates so that at least one of the servers is available. When redundant processes (either via simple replication of code or via multiple versions) are used to achieve a result, a voting scheme is usually employed to decide on the output that must be actually produced.

Reconfiguration can be performed to recover from faulty processors by replicating the process to an operational processor, or to replace a malfunctioning version with a correct one. A model to dynamically reconfigure software in distributed systems is presented in [9]. It analyzes the dependencies among the different processes in the system to determine the impact of a given reconfiguration operation. The model has been used to build a fault-tolerant environment based on ARMOR processes (Adaptive Reconfigurable Mobile Objects of Reliability). However the model itself does not advise on which kind of reconfiguration should be performed under which circumstances. It leaves this task to system administrators or possibly other software components with the ability to make such decisions. Moreover, if the checkpointing or replication logic becomes corrupted, fault tolerance will not be achievable anymore. We call this an extrinsic approach where fault tolerance logic is incorporated into a system as an add on. Our interest is in an intrinsic solution where the robustness is part of the (protocol) software itself.

2.3 Self-testing and self-correcting software

The usual methodologies for software testing and debugging rely on running the program under a controlled environment using a subset of possible inputs. The corresponding outputs must be known beforehand, and in most cases bugs arise after the software is deployed, due to untested combinations of input data or events.

Self-testing and self-correcting programs [10, 11] have been proposed to improve the reliability of software systems. Given a program P, a self-testing program for P is another, simpler program designed to make calls to P on a number of inputs, and to check whether the corresponding outputs are correct. A self-corrector for the same program P attempts to return the correct output for each input value, even in the presence of abnormal behavior of P, provided that the probability of P producing the wrong output is small enough.

While these techniques can in principle be useful for software debugging and testing, and to improve software reliability at run time, in practice especially the self-correcting functions can be hard to design, since they are very specific for each program. Another shortcoming of self-correcting functions is that they do not correct the program itself, but just attempt to correct wrong outputs.

In [12] the authors extend the notion of self-testing/correcting functions to distributed processing or protocols. Their approach has the same advantages and limitations of [10, 11], but applied to protocols in distributed systems, which makes the testing and correcting algorithms more complex.

Again, self-testing or correction is an extrinsic approach that works on a given program: it does not apply to self-testing and correcting its own operation.

2.4 Unfaithful Mobile Code Execution and its Detection

The "malicious host problem" refers to an execution environment which actively tries to distort a program's execution or to extract valuable data from it. This problem, which has been extensively discussed in the mobile agent community, is closely related to the setting discussed here where we assume (one) random execution errors or errors from unknown sources. In the examples presented in section 4 we are pursuing an approach where robustness can be obtained through protocol transformations "in the clear". However, an encoded (or, as in [13], an "encrypted") execution is also conceivable and would rejoin the approach taken in quantum computing, as described below.

The detection of incorrect execution also belongs to the context of malicious hosts and our quest for resilient protocols. In [14] a watermark is added to the data which permits to verify whether a remote operation was duly executed. This is structurally similar to the technique used further down in an example where each operation has side effects which serve as hooks for detecting the malfunctioning of an execution circuit.

2.5 Fault Tolerant Quantum Computing

A key aspect of quantum computing is that quantum state (e.g. in a qubit) should not interact with its environment in an uncontrolled way. Unlike classical computers, where state can be measured, restored and copied, it is not possible to copy quantum state or to remove inevitable noise from quantum operations by some threshold scheme. However, it was shown that with an appropriate encoding of a qubit as a codeword over several qubits, it is possible to implement error *detection* capabilities (a) with quantum operations, (b) such that the detection circuitry can also be subject to potential errors and (c) that errors can be *corrected* – again with quantum operations – permitting arbitrarily long quantum computations [15].

This corresponds exactly to what we aim at in this paper: Our goal is to obtain intrinsically robust protocol implementations where errors can occur in the "core" protocol implementation *as well as* in the detection and correction part. However, one difference to the fault tolerant quantum computing approach is that we do not target a logical gate level abstraction on top of which the full (communication) software hierarchy could be stacked. Instead, we wish to expose the unreliability of execution to the highlevel protocol description and handle it inside the protocol. That is, we do not want to (re-)implement basic logic gates and construct computation networks out of them as quantum computing theory does. Another important difference is that we are interested in *software*, and in *arbitrary interconnections between code pieces*, instead of fixed hardware circuits.

2.6 Resilience and self-healing properties of biological systems

Emergent behavior in biological systems leads to self-regulatory feedback systems that are robust to external perturbation and to failure of its constituent parts [16]. These systems tend to be highly decentralized, the emergent behavior resulting from simple interactions among autonomous agents that make decisions based solely on local views. These simple agents are often anonymous and non-specific, leading to intrinsic fault tolerance and self-healing properties, as other agents can easily take over the roles of failed or missing ones. Several examples can be cited, such as evolutionary selection with survival of the fittest, flock of birds, colonies of social insects, etc. In this section we concentrate on the biochemical processes occurring in cells, which inspired the Fraglets paradigm.

Within a living cell or microorganism there are several chemical processes responsible for maintaining the various cellular functions. These processes can be represented by graphs (networks) where the nodes are the chemical compounds and the links are the reactions that transform these compounds. Biochemical pathways describe the sequence of chemical reactions inside a biochemical network. Among the numerous cellular biochemical networks we can distinguish metabolic networks, responsible for the cell's energy cycle.

These cellular processes are known to be highly robust against mutations, due to several redundancy and diversity mechanisms, such as [17]: the presence of multiple genes with similar functions; interactions among genes with unrelated functions, such as in the case of recessive mutations; the existence of alternative routes in large metabolic networks; the scale-free nature of metabolic networks [18], which are dominated by a few highly connected hubs, while the vast majority of the nodes have a small number of connections, making them inherently robust to random errors.

2.7 Core Wars

Finally we mention "core wars" [19]: Two programs, which share the same random access (core) memory, struggle for survival by attacking the other program through tampering with its instructions and/or by evading attacks through dislocation. Various robustness and self-healing strategies have been proposed for this rather specific context and the associated virtual machine.

3 Parallel and Dynamic Execution Models for Protocols

The traditional implementation of communication protocols with sequential processing of instructions is hard if not impossible to "robustify": An error in a single instruction will most likely disrupt the fragile execution path. Instead, we seek an execution environment where several fine-grained activities can go on in parallel and which can serve as a backup. In this section we present three systems which permit a more parallel expression of communication software.

3.1 Gamma

Back in 1986, Banâtre and Métayer proposed Gamma [20], a programming formalism based on a chemical reaction metaphor. A good overview of the topic with its many ramifications can be found in [21], and a recent update in [22]. Gamma computations consist in "chemical reactions" which consume elements of a multiset data structure, and produce new elements to the multiset. This model enables highly parallel programs to be expressed in a way that is very close to their specification. The authors show that this property makes gamma systems particularly suited as a basis for automated program synthesis.

Many extensions and variations of the basic Gamma system have been proposed, for instance, the Chemical Abstract Machine (CHAM) [23] and Membrane or P systems [24].

These chemical execution models have been applied to diverse fields [21] such as image processing applications, operating systems, compilers, dynamic software reconfiguration [25], multi-agent systems [26] and distributed computing [27]. More recently, γ-calculus has been introduced as a formalism that extends the original Gamma model to a higher-order calculus. In [28] this new calculus has been applied to specify Autonomic Computing systems, including a mailing system as an example. However, to the best of our knowledge, such models have not yet been used to create or reconfigure network protocols.

3.2 Communicating Rule Systems

A formal framework that explicitly addresses communication software is the *Communicating Rule Systems (CRS)* by Mackert and Mackert [29]. Basically a condition/event type of a system, it potentially permits to capture execution variety at the level of single rules such that alternative rules could take over should another rule become unavailable. The rule base, however, is static (due to the author's interest in protocol validation) and becomes a limitation when we want to modify or restore a protocol implementation at run time.

3.3 The Fraglet Paradigm

The Fraglet paradigm [3] has been proposed as part of our search for feasible ways to achieve automated synthesis of protocol implementations. It is based on a chemical model where "molecules" interact with each other or undergo some internal transformation. Formally, it is an instance of Gamma systems [20, 22], described above. Like the higher-order γ-calculus [22, 28], Fraglets explicitly represent code and data in a unified way. The code itself is part of the multiset, a metaphor which is even closer to real chemical systems when compared to the original Gamma model [20, 21]. Adopting this specific chemical model (with Fraglets as the only objects) has the benefit of being able to integrate code deployment into protocols in a natural way. As we will see in Section 4, this will also facilitate the production of instruction-failure resilient code.

A fraglet is a string of symbols $[\, s_1 : s_2 : \ldots : s_n \,]$ which represents data and/or protocol logic. It represents, so to speak, a fragment of a distributed computation. Fraglets may reside inside a node's fraglet store or may be carried in packets, where successive fraglet symbols are analogous to successive header fields in today's regular data packets. Upon arrival, a fraglet packet is injected into the local fraglet store or context.

The fraglet processing engine continuously executes tag matching operations on the fraglets in the store, in order to determine the actions that should be applied to them. Fraglet operations, except for the transmission, have the property that they can be carried out in constant time.

Formally, the store is a multiset: several instances of the same fraglet may be simultaneously present. This is indicated by a suffix counter value as in $[\, data : item \,]k$ (meaning that k copies of fraglet $[\, data : item \,]$ are stored in this context).

The fraglet instruction set currently contains two types of actions: transformation of a single fraglet, and reaction between two fraglets. Table 1 shows some transformation rules defined so far. Table 2 shows the reaction rules.

Table 1. Transformation rules

Op	*Input*	*Output*
dup	$[\,dup : t : u : tail\,]$	$[\,t : u : u : tail\,]$
exch	$[\,exch : t : u : v : tail\,]$	$[\,t : v : u : tail\,]$
new	$[\,new : t : tail\,]$	$[\,t : n_{i+1} : tail\,]$
split	$[\,split : t : \ldots\ : * : tail\,]$	$[\,t : \ldots\,]$, $[\,tail\,]$
send	$_A[\,send : B : tail\,]$	$_B[\,tail\,]$ (unreliably)

The semantics of the transformation rules are:

– *dup*: Duplicates the symbol at the third position (u); the second field (t) becomes the new fraglet's head symbol.
– *exch*: Swaps the symbols at the 3rd and 4th position (u and v respectively).
– *new*: Creates a new symbol n_{i+1} which is unique in this context.
– *split*: Breaks the fraglet into two parts at the first marker position ($*$).
– *send*: Sends the fraglet unreliably to the destination context specified by the second symbol (B). The subscript prefix $_N[\ldots]$ denotes the context where the fraglet is stored.

Table 2. Reaction rules

Op	*Input*	*Output*
match (merge)	$[\,match : s : tail_1\,]$, $[\,s : tail_2\,]$	$[\,tail_1 : tail_2\,]$
matchp (persist)	$[\,matchp : s : tail_1\,]$, $[\,s : tail_2\,]$	$[\,tail_1 : tail_2\,]$ $[\,matchP : s : tail_1\,]$

We introduced two simple reaction rules, listed in Table 2. The "merge" instruction (*match*) concatenates two fraglets with matching tags. The "persist" variant (*matchP*) moreover keeps a copy of the initial $[\,matchP : \ldots]$ fraglet in the store, thus acts like a catalyst.

It is important to emphasize that our research in defining the instruction set is still in progress, thus the current state should be interpreted as a snapshot of an evolving work. In spite of this apparent limitation, in [3] two protocol implementations using fraglets have been shown: a very simple confirmed delivery protocol and a more complex flow control protocol with send credit and packet reordering.

4 Resilient Protocols

In this section we demonstrate a few simple computation tasks in a communication context whose implementation is robust against partial erasure of their code base. We start with our definition of protocol robustness and instruction failure, and a discussion of appropriate languages to achieve robustness to instruction failure. Then we show two examples of resilient programs at the instruction level: a "signaling frequency doubler" and a confirmed delivery protocol. We then discuss the resulting code and insights gained from this exercise.

4.1 Robustness

Today we have a methodology of designing protocols that makes rather strong assumptions on the reliability of the executing components. The single execution environments are presumed to be stable: Reliable protocols that execute on these components are supposed to handle "only" the unreliable aspects of networking, like broken links, lost packets or transmission errors. That is, robustness applies to the harsh (communication) environment in which computers operate, not to the execution support itself.

Here, we redefine protocol resilience, or protocol robustness, as the ability to survive instruction failures: Even when portions of the protocol's implementation are lost (or duplicated, or changed), the protocol is still able to provide the intended service, albeit with some loss of efficiency. Ideally, a robust protocol implementation will not only be able to detect but also to recover from code losses (self-healing).

Our definition of "protocol robustness" is linked to an implementation, and therefore to the instruction set it is based on. At this point, a question to be raised is which kind of programming languages are suitable to produce instruction-level failure resilient code. Classical procedural languages such as C and Java are very poor candidates, since each single instruction (e.g. assignment, if-then-else, etc.) is linked to many others via cause-effect relationships. Erase a single assignment and the program is likely to crash entirely. Redundancy cannot be directly applied here.

In this paper we use the Fraglet system as our instruction set and consider examples where redundancy can be applied such that any fraglet rule (which is a fraglet by itself) can be removed without changing the outcome of the computation.

An important property that makes fraglets more suitable to instruction-failure resilience is the integration of code and data into the multiset pool. This enables redundancy of code to be expressed in a natural way, without harmful side effects: among a set of redundant rules that match a given input stream, only one of them will be chosen. If at least one of the rules is present, the program can run smoothly in spite of its other sibling rules being knocked out. This will become more clear with the examples of sections 4.2 and 4.3.

4.2 A Robust Message Doubler with Fraglets

The simple task we want to solve is the doubling of a signal stream: for each message x we want two messages z to leave a node or to be available for further processing inside that node. For instance, the messages x could be the ticks from a Geiger counter sensor or any other source which encodes information as a frequency.

Rewriting the signal x (represented as a Fraglet $[x]$) to the new name can be done by the single (non-robust) fraglet rule

$[\ matchp : x : z\ ]$

where the $[x]$ will be replaced by $[z]$. The doubling is done by a slightly larger rule, namely

$[\ matchp : x : split : z : * : z\ ]$

which says that fraglet $[x]$ is replaced by a $[\,split...]$ fraglet. This new fraglet will break apart in the next processing step and produce two $[z]$ Fraglets, which is the desired effect.

Unfortunately, this program is not robust: Removing the rule will effectively erase the program. A trivial way to obtain robustness in the Fraglet is to double the rules:

$[\ matchp : x : split : z : * : z\ ]$
$[\ matchp : x : split : z : * : z\ ]$

Here, any one rule instance can be removed, leading to the other rule to be invoked twice as much as before. This seems to satisfy the resilience condition: However, it is not possible to determine, from the program's output, whether a fraglet (rule) has been lost or not. After the first loss, the program is no longer resilient, although it continues to run and produce results, until the single remaining copy is also lost. In order to achieve long-term resilience, the loss of the first Fraglet must be signaled, permitting to trigger a self-healing process.

Therefore we need an implementation such that the side effect of a loss is not harmful to its intended core functionality, but permits to identify which fraglet rule became unavailable, operating as a signal that can be used as input for a self-healing process.

The next version of the doubling program, also shown in Figure 1, is:

Fraglet rule 1: $[\ matchp : x : split : s1 : * : s1\ ]$
2: $[\ matchp : x : split : s2 : * : s2\ ]$
3: $[\ matchp : s1 : split : z : * : a\ ]$
4: $[\ matchp : s1 : split : z : * : b\ ]$
5: $[\ matchp : s2 : split : z : * : c\ ]$
6: $[\ matchp : s2 : split : z : * : d\ ]$

In this case, we have *two different* rules that transform an input fraglet x into a *split* fraglet. This means that with a 50% chance, one of these two rules will be picked. Depending on which rule was picked, either two $[\,s1 : \ldots]$ or two $[\,s2 : \ldots]$ fraglets will be produced. At this level we have again two rules which transform an $[\,s1 : \ldots]$ into two different $[\,split : \ldots]$ fraglets. In fact, it is at this level that the doubling of the original messages is happening: half of the $[x]$ fraglets will become $[\,s1 : \ldots]$ fraglets, but because of the two levels and branches

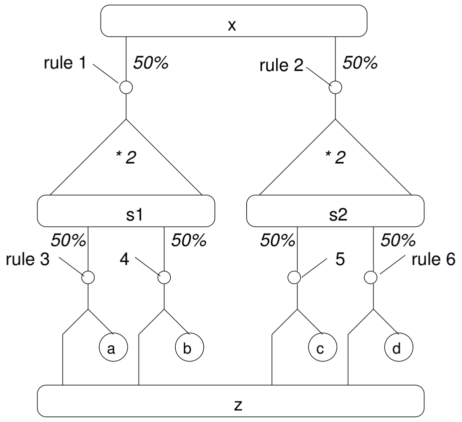

Fig. 1. The processing of fraglets for the robust doubler program.

of $[\,split : \ldots]$ (resulting in a quadrupling), we have an overall doubling of the incoming flux of $[x]$ fraglets.

One can observe that removing any one of these 6 fraglet rules will not change the net outcome. If the first rule is removed, the second rule will react with 100% of the influx: for each $[x]$ fraglet there will be two $[\,s2 : \ldots]$ fraglets which each produce a $[z]$ fraglet as desired. Similarly, removing any of the other rules will just lead to its "homologue" rule to take over all intermediate fraglets instead of processing only 50% of them.

Note that this implementation generates additional "side effects" in form of the fraglets named $[a]$, $[b]$ etc. These signals can be used to react on the loss of a rule, as is further discussed in section 4.4.

4.3 A Robust Confirmed Delivery Protocol

This example is a simple confirmed delivery protocol (CDP) already shown in [3]. Node A sends an input *data* fraglet to node B; when B receives the data it delivers it to the application and returns an *ack* fraglet to node A. One possible implementation of this protocol, in its non-resilient form, is:

$_A\ [\ matchp : cdp : send : B : deliver\ ]$
$_B\ [\ matchp : deliver : split : send : A : ack : *\ ]$

The resilient version can be obtained by simply applying the same technique used for the doubler, that is, duplicate the rule and amend it to generate a unique symbol that allows to identify which rule has been executed. The resulting protocol becomes:

$_A\ [\ matchp : cdp : split : a : * : send : B : deliver\ ]$
$_A\ [\ matchp : cdp : split : b : * : send : B : deliver\ ]$
$_B\ [\ matchp : deliver : split : c : * : split : send : A : ack : *\ ]$
$_B\ [\ matchp : deliver : split : d : * : split : send : A : ack : *\ ]$

Actually it is possible to apply the same transformation to any *matchp* rule in order to make it robust. This is a straightforward way to build resilient programs out of normal ones.

4.4 Discussion

The main “trick” of our demo programs is the doubling of rules which compete against each other. This so called “soup” aspect of chemical execution model makes it much easier to write robust code which continues operation despite some losses: any sequential execution model where the control flow must pass through a single instruction is potentially not amendable to robustness, because there is no fallback execution path.

Self-Healing: An important aspect of our program is that it produces a stream of signalling symbols, the fraglets a, b, c and d in the examples. These can be used to monitor the health of the program and to trigger repair mechanisms. Adding a few “cleaning” rules will remove these signal streams in case everything works well. For the doubler program, the cleaning rules are:

$[\ matchp : a : match : c\ ]$
$[\ matchp : b : match : d\ ]$

And for the CDP program, they are:

$_A\ [\ matchp : a : match : b\ ]$
$_A\ [\ matchp : b : match : a\ ]$
$_B\ [\ matchp : c : match : d\ ]$
$_B\ [\ matchp : d : match : c\ ]$

As soon as one of the Fraglet rules is removed, this balance is disturbed: By looking at the relative weights of debris ($[a]$, $[\,match : a\,]$ etc.), one can infer for almost all cases which rule was removed. We note here that despite all these detection activities, the doubler program continues to produce twice as many $[z]$ as $[x]$ Fraglets, and CDP continues its normal flow of data and acknowledgments.

The design of control loops – which for example regenerate a lost $[\,matchp : \ldots]$ rule – is not trivial and, as our first experiments show, will probably lead to solutions where perfect robustness is not achievable anymore because the result stream can show temporary distortions. Nevertheless, the resulting program performs in an elastic way without fully disrupting the processing. Potentially, one should be able to produce programs which regain correct status after the healing process and given that the error rate is not too high.

However, the exact way to produce healing code able to reconstruct the lost rule is still work in progress. One possibility is to copy one of the remaining rules and rewrite its unique identifier symbol to produce another redundant rule. Another issue is that ideally, the healing code itself should be resilient. Therefore it does not suffice to create a healing “meta-level” that regenerates lost fraglets: a truly self-healing solution should be self-contained. Such a self-contained solution could be obtained in two ways: either by rewriting every program to become resilient and self-healing in itself, or by finding a single "healer" program that is self-healing by itself, i.e. without relying on a meta-level or on special platform features. This program could be used to repair other programs to achieve full self-healing ability, without leading to infinite regression.

Resource Control: In a fully self-healing process we have to deal with resource control issues: unlike the simple demo examples above, which react on the *loss* of fraglet rules, one also has to anticipate an *excess* of rules. Instead of introducing logic to remove superfluous code, we consider the continuous generation of code on the fly and on demand, all new code being automatically consumed after it has been processed. This would lead to a completely dynamic program where the $[\,matchp : \ldots]$ rules would be replaced by $[\,match : \ldots]$ rules. These later fraglets would have to be constantly regenerated, thus leading to a program that would be constantly rewriting itself according to its own sensed performance. Metaphorically, we need transcription signals which control the gene expression.

Distribution: As mentioned in the introduction, resilience can be put in a communication context. By changing the intermediate fraglet types such that they undergo a "transmission"-transformation, one can have different parts of the process to occur on different nodes. For the doubler example, assuming that we have four nodes N, M, O and P, we would have the rules:

$_N\ [\ matchp : x : send : M : split : s1 : * : s1\ ]$
$_N\ [\ matchp : x : send : O : split : s2 : * : s2\ ]$

$_M\ [\ matchp : s1 : split : send : P : z : * : send : P : a\ ]$
$_M\ [\ matchp : s1 : split : send : P : z : * : send : P : b\ ]$

$_O\ [\ matchp : s2 : split : send : P : z : * : send : P : c\ ]$
$_O\ [\ matchp : s2 : split : send : P : z : * : send : P : d\ ]$

$_P\ [\ matchp : a : match : c\ ]$
$_P\ [\ matchp : b : match : d\ ]$

From node N we rewrite $[x]$ fraglets into fraglets that transfer themselves to nodes M and O and split there. On these nodes, the doubling is performed and the resulting stream is redirected to node P. Note that here we start to blur the distinction between traditional robustness protocols, as the crash of a node M (resulting in removing two rules) would be observable by an imbalance of the monitoring signals and could trigger the necessary repair actions.

5 Conclusions and Outlook

In this paper we have demonstrated simple programs that continue to perform their task despite the removal of any of their instructions. This "intrinsic" robustness is different from the usual extrinsic fault-tolerance approaches as it weaves self-monitoring and self-healing into the proper processing.

The ultimate goal is to create software for autonomic networks which is resilient to accidental or malicious code manipulation and execution problems. In this paper we have shown only the first step towards this goal: instruction-failure resilience, which is achieved using a "chemical" protocol representation and execution model where fallback actions and action equilibria can be easily expressed. Our examples were based on the Fraglet instruction set and multiset memory which has the same fine grained parallelism as the formal Gamma model.

The list of potential research topics in this area is immense. Resilience should be extended to a more general scope beyond single instruction failure: It should be possible to apply the insights from single-instruction knock-out experiments to *networks of components* where medium sized software elements inside a node can crash in a globally recoverable way, or where the amount of elements running concurrently inside the whole network must be controlled. Another important issue is a methodology for transforming existing protocols into a robust, self-healing implementation.

This latter aspect is still work in progress, namely to generate self-healing implementations able to react on the anomaly signals produced by the resilient code, and to regenerate the code base according to these signals. Realistic synthesis methodologies for intrinsically robust protocols will certainly take a long time to mature, even more for self-healing protocols where more insights are needed into "code dynamics".

References

1. Smirnov, M.: Autonomic Communication: Research Agenda for a New Communication Paradigm. White paper, Fraunhofer FOKUS (2003)
2. Zambonelli, F., Gleizes, M.P., Mamei, M., Tolksdorf, R.: Spray Computers: Frontiers of Self-Organization. In: Proceeding of 1st International Conference on Autonomic Computing (ICAC'04), New York, USA (2004) 268–269
3. Tschudin, C.: Fraglets - a Metabolistic Execution Model for Communication Protocols. In: Proceeding of 2nd Annual Symposium on Autonomous Intelligent Networks and Systems (AINS), Menlo Park, USA (2003)
4. Andersen, D.G., Balakrishnan, H., Kaashoek, M.F., Morris, R.: Resilient Overlay Networks. In: Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP 2001), Banff, Canada (2001)
5. Jalote, P.: Fault Tolerance in Distributed Systems. ISBN 0133013677. Pearson Education (1994)
6. Dialani, V., Miles, S., Moreau, L., Roure, D.D., Luck, M.: Transparent Fault Tolerance for Web Services based Architectures. In: Proceedings of 8th International Europar Conference (EURO-PAR'02), Paderborn, Germany (2002) 889–898
7. Mullender (Ed.), S.: Distributed Systems, Second Edition. ACM Press, Addison-Wesley (1993)
8. Torres-Pomales, W.: Software Fault Tolerance: A Tutorial. Technical Report TM-2000-210616, NASA (2000)

9. Whisnant, K., Kalbarczyk, Z.T., Iyer, R.K.: A system model for dynamically reconfigurable software. IBM Systems Journal **42** (2003) 45–59
10. Blum, M., Luby, M., Rubinfeld, R.: Self-Testing/Correcting with Applications to Numerical Problems. Journal of Computer and System Sciences **47** (1993) 549–595
11. Wasserman, H., Blum, M.: Software Reliability via Run-Time Result-Checking. Journal of the ACM (JACM) **44** (1997) 826–849
12. Franklin, M., Garay, J., Yung, M.: Self-Testing/Correcting Protocols. In Jayanti, P., ed.: 7th International IS&N Conference on Intelligence in Services and Networks (ISN'00). Springer-Verlag LNCS 1693, Bratislava (1999) 269–283
13. Sander, T., Tschudin, C.: Towards mobile cryptography. In: Proceedings of the IEEE Symposium on Security and Privacy, Oakland, CA, USA, IEEE Computer Society Press (1998)
14. Sander, T., Tschudin, C.F.: On software protection via function hiding. Lecture Notes in Computer Science **1525** (1998) 111–123
15. Preskill, J.: Fault-Tolerant Quantum Computation (1997) arXiv:quant-ph/9712048.
16. Anthony, R.J.: Emergence: A Paradigm for Robust and Scalable Distributed Applications. In: International Conference on Autonomic Computing (ICAC-04), New York, USA (2004)
17. Fontana, W., Wagner, A.: Mutational Robustness, Modularity and Evolvability. Research focus area robustness, Santa Fe Institute (2002) `http://www.santafe.edu/sfi/research/focus/robustness/projects/mutationa%lRobustness.html`.
18. Jeong, H., Tombor, B., Albert, R., Oltvai, Z., Barabási, A.L.: The large-scale organization of metabolic networks. Nature **407** (2000) 651–654
19. Dewdney, A.K.: Recreational Mathematics – Core Wars (May 1984) Scientific American. See also `http://www.koth.org/`.
20. Banâtre, J.P., Métayer, D.L.: A new computational model and its discipline of programming (1986) Technical Report RR0566, INRIA.
21. Banâtre, J.P., Métayer, D.L.: Gamma and the chemical reaction model. Internal publication pi-984, INRIA (1996)
22. Banâtre, J.P., Fradet, P., Radenac, Y.: Principles of chemical programming (2004) Fifth International Workshop on Rule-Based Programming (RULE'04).
23. Berry, G., Boudol, G.: The chemical abstract machine. Research report rr-1133, INRIA Sophia Antipolis (1989)
24. Paun, G.: Computing with Membranes. Journal of Computer and System Sciences **61** (2000) 108–143
25. Wermelinger, M.A.: Specification of Software Architecture Reconfiguration. PhD dissertation, Universidade Nova de Lisboa, Lisbon, Portugal (1999)
26. I. Stamatopoulou, M. Gheorghe, P.K.: Modelling of dynamic configuration of biology-inspired multi-agent systems with communicating X-machines and population P systems. In: Fifth Workshop on Membrane Computing (WMC5), Milan, Italy (2004)
27. Syropoulos, A.: On P systems and distributed computing. In: Fifth Workshop on Membrane Computing (WMC5), Milan, Italy (2004)
28. Banâtre, J.P., Radenac, Y., Fradet, P.: Chemical specification of autonomic systems. In: Proc 13th International Conference on Intelligent and Adaptive Systems and Software Engineering (IASSE'04). (2004) 72–79
29. Mackert, L.F., Neumeier-Mackert, I.B.: Communicating Rule Systems. Protocol Specification, Testing and Verification VII, Proc IFIP WG6.1, 7th International Conference on Protocol Specification, Testing and Verification, Zurich, Switzerland (1987)