A Self-Healing Load Balancing Protocol and Implementation

Thomas Meyer and Christian Tschudin

Technical Report CS-2009-001
University of Basel
July 29th, 2009

Abstract

Beyond the mere collection of computers, a network is the home of competing and cooperating execution flows. In this paper we show how to design network protocols based on molecule-like entities such that the corresponding execution flows can by analyzed as if they were chemical processes. Our goal is to create *robust protocol implementations* which are resilient to unreliable execution. We introduce the metaphor of chemical networking protocols, demonstrate its benefits by a formal stability analysis of a gossip-style protocol and present a first example of a self-healing load balancing protocol that is resilient to code removal attacks.

Keywords:
CNP, protocol implementation, protocol analysis, system design, unconventional computing.

Computer Science Department, Computer Networks Group, Bernoullistrasse 16, CH – 4056 Basel Switzerland. Phone: +41 61 267 05 55 Fax: +41 61 267 05 59 URL: http://cn.cs.unibas.ch/

1 Introduction

Every now and then, network engineers are making use of chemical terms to describe properties of their protocols: TCP's congestion control algorithm bases on the "conservation of packets principle" trying to bring a connection to "equilibrium" [16]. AntNets stochastically routes packets proportional to "pheromone concentrations" [7]. Gossip-style protocols like [18] adopt the dynamics of epidemic spreading to disseminate information in a robust way. Apparently, chemically inspired ideas are considered to be beneficial for the dynamic behavior of protocols.

For the sake of stimulating the discussion, we are taking the extreme position of using chemistry as the central dogma for designing protocols and explore the potential of this approach. We aim at transposing properties of chemical systems to protocols that are hard to obtain otherwise. This includes autonomously finding equilibria at the point of optimal operation or the protocol's robustness to internal and external perturbations.

We introduce a programming model to describe computation as well as network communication as abstract chemical reactions and show two examples of "Chemical Networking Protocols" (CNPs). The first example "chemically calculates" the average of distributed values. Because of the analogy to chemical reaction networks, we can make use of analytical tools developed over decades in chemistry to predict the behavior of such systems, like for example Metabolic Control Analysis [15] or Chemical Organization Theory [8]. We will give an example for this analysis potential by formally proving the protocol's stability.

In the second part of this paper we ask whether some higher-level properties, such as the capability of biological systems to heal themselves, can be achieved in our model. This is important with regard to future execution environments that may be unreliable, such as probabilistic chips [5] or deep space probes, where the question is how to write communication software that tolerates spurious execution errors. In this paper we show how to construct self-replicating molecules that bring forward self-healing software. We apply this to CNPs and demonstrate a path load balancing protocol that is robust not only to the loss of packets but also to the deletion of code.

2 Chemical Reaction Model

Traditionally, protocol execution is handled by a state machine that upon the reception of a packet synchronously changes its internal state and performs some communication activity. Here we introduce a "molecule metaphor" where each packet is treated as a virtual molecule. Virtual molecules react with other molecules in a reaction vessel (node). A reaction may produce other molecules being delivered to the application or being sent over the network. In such a chemical perspective, we obtain a web of reactions that together perform a distributed computation (called network service).

2.1 Modeling Chemical Communication

Instead of encoding a deterministic state machine, or having a sequential program that processes an incoming packet, each network node contains a multiset $\mathcal{M}(\mathcal{S})$ of a finite set of molecules $\mathcal{S} = \{s_1, \ldots, s_n\}$ (=packets). In addition, each node defines a set of reaction rules $\mathcal{R} = \{r_1, \ldots, r_m\}$ expressing which reactant molecules can collide and which molecules are generated during this process. Such a reaction is typically represented as follows:

$$i: \quad C_i + X_i \longrightarrow C_i + X_j \tag{1}$$

The above reaction in node i consumes, if present, two molecules C and X from the local multiset, regenerates C and sends molecule X to neighbor node j. In a simple two-node network topology, the above example spans the following reaction network, also depicted in Fig. 1:

$$1 : C_1 + X_1 \xrightarrow{r_1} C_1 + X_2 \quad (2)$$

$$2 : C_2 + X_2 \xrightarrow{r_2} C_2 + X_1 \quad (3)$$

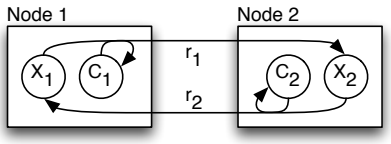

Figure 1: Distributed reaction network spanned by two identical local rules, (2) and (3), resp.

A received molecule is in a first step passively placed into the multiset of the node. For example, rule (3) is not executed immediately after node 1 receives a new X molecule. It is rather scheduled for a later time determined by an exact stochastic reaction algorithm, such as [14, 13]. The role of this delayed execution is to enforce the "law of mass action" at the macroscopic level. In chemistry, the "law of mass action" states that the reaction rate is proportional to the concentration of reactant molecules [3]: Molecules C_1 and X_1 react with an average rate equal to the product of their concentration $r_1 = c_1 x_1$. The rate of packets sent from node 1 to node 2 is equal to r_1 while the packet stream in the opposite direction exhibits a rate of $r_2 = c_2 x_2$. It can be shown that the overall reaction system spanned by the two local reaction rules strives towards the equilibrium where the number of X molecules in either node is inversely proportional to the number of the corresponding C molecules.

2.2 Representation-free Encoding

Unlike traditional protocols, our chemical networking protocols use a representation-free encoding of information. Traditional protocols store their local state symbolically[1] in variables, such as integers or flags, ultimately encoded as bit patterns. Such symbolic information is then piggybacked to packets in order to send information to distant nodes. CNPs encode protocol states as concentrations, here the abundance of molecule X in the multiset. This encoding is more robust: When removing a molecule, the concentration only changes marginally and state information is not lost, only disturbed.

Due to the law of mass action, the packet *rate* reflects the concentration of its originating chemicals (e.g. $r_1 \propto x_1$). Thus, the packet rate itself, not the symbolic information inside the packet, is used to communicate state information among nodes.[2] This rate coding scheme akin to the nervous system [6] results in a higher resilience to the loss of packets. The law of mass action plays an important role: It mediates between the local and global world by proportionally mapping molecule concentrations to packet rates and vice-versa.

3 Computing Aggregates as Chemical Equilibria

In this section we present an example of a chemical networking protocol (CNP) that calculates the average of distributed values. We compare our chemical approach to epidemic protocols.

[1] Symbol = meaningful task-related reduction of information [4].

[2] If more than one molecule type is exchanged, however, we have to pass the identity of the molecule along with the packet to distinguish them at the receiver's side.

3.1 The Gossip-style *Push-Sum* Protocol

Recently, gossip-based or epidemic protocols gained attention because of their potential to disseminate information in a robust way [12]. For example, the *Push-Sum* protocol [18] averages out locally stored values by means of a simple local algorithm: Node i stores sum and weight as tuple (s_i, w_i), starting with $(x_{i0}, 1)$ where x_{i0} is the node's initial (sensor) value. In each round, i.e. after a fixed time interval, each node i first sends the tuple $\left(\frac{1}{2}s_i, \frac{1}{2}w_i\right)$ to a randomly chosen neighbor and to itself and collects the tuples $\{(s_{ir}, w_{ir})\}$ received from neighbors in this round. Then it sums up the received values $s_i := \sum_r s_{ir}$ and $w_i := \sum_r w_{ir}$. Like this, the fraction $x_i = s_i/w_i$ asymptotically approaches the average over all values of the network. Although the protocol is simple, the "proof of the approximation guarantee is non-trivial" [18].

3.2 A Chemical *Disperser* Protocol

In the following we present an alternative, chemical implementation that is elegant and intuitively graspable. It also greatly simplifies the analysis such that the convergence proof keeps on roughly half a page. We use the usual network representation as undirected graph $G = (V, E)$ where V is the set of $|V|$ vertices (nodes), E is the set of edges (links). $A = [a_{ik}]$ is the adjacency matrix of the graph where $a_{ik} = a_{ki} = 1$ if $(i, k) \in E$, or $a_{ik} = 0$, otherwise.

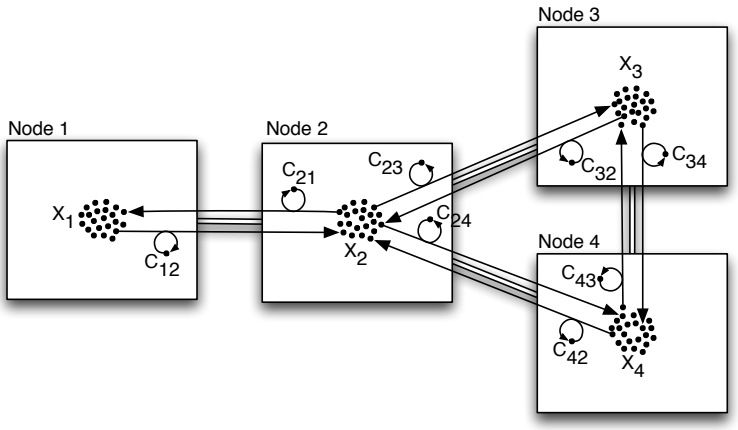

Figure 2: The chemical *Disperser* protocol calculates the average concentration of X molecules in a network of nodes.

Figure 2 schematically depicts the components of the chemical *Disperser* protocol. Each node $i \in V$ contains a multiset of the following molecule types: The concentration of X_i represents the computed average, which is initially set to the local value x_{i0}. For each link $(i, k) \in E$ to i's neighbors there is a single instance of molecule C_{ik} that reacts with the local X_i molecule and, by doing so, sends the X to the corresponding neighbor node k. Formally, such a reaction is represented by the following chemical equation:

$$C_{ik} + X_i \xrightarrow{r_i} C_{ik} + X_k \qquad \forall (i, k) \in E \tag{4}$$

We can intuitively feel that the global reaction network should lead to equilibrium: The more neighbors a node has, the greater is the outflow of X, since there are more C molecules to react with. On the other hand, a node with higher degree also receives X molecules from more neighbors.

Formal Convergence Proof

To formally prove the stability of the chemical algorithm, we perform a perturbation analysis at the fixpoint. Therefore we first write down the differential equation for the concentration of X_i:

$$\dot{x_i} = \overbrace{\sum_{k \in V} a_{ki} x_k c_{ki}}^{\text{inflow}} - \overbrace{\sum_{k \in V} a_{ik} x_i c_{ik}}^{\text{outflow}} \qquad \forall i \in V \tag{5}$$

To find a fixpoint we set $\dot{x_i} = 0$. Since there is only one control molecule C_{ik} per link we simplify (5) by setting $c_{ik} = a_{ik}$. Solving this equation w.r.t. x_i yields

$$x_i = \frac{\sum_{k \in V} a_{ki} x_k}{\deg(i)} \qquad \forall i \in V \tag{6}$$

Hence, x_i is equal to the average concentration of X in i's neighbors, which only holds iff

$$x_i = \frac{\sum_{k \in V} x_k}{|V|} = \langle x \rangle \qquad \forall i \in V \tag{7}$$

Consequently, at chemical equilibrium the X molecules are equally distributed over the network.

This equilibrium is stable if the system returns back to the fixpoint after a small perturbation. For the corresponding analysis, we calculate the $|V| \times |V|$ Jacobian matrix of (5):

$$J = [j_{ik}] = \left[\frac{\partial x_i}{\partial x_k}\right] = -L(G) \tag{8}$$

where $L(G)$ is the Laplacian of the network graph

$$L(G) = [l_{ik}] = \begin{cases} \deg(i) & \text{if } i = k, \\ -a_{ik} & \text{otherwise.} \end{cases} \tag{9}$$

Since any Laplacian has positive real eigenvalues, the eigenvalues of (8) are negative and the fixpoint in (7) is asymptotically stable for arbitrary network topologies.

3.3 Protocol Comparison

In order to illustrate the principle of the two protocols, we carried out OMNeT++ [2] simulations in a simple four node topology (see Fig. 2).[3] For *PushSum*, we used an update interval of 250 ms. Both protocols start with an initial value of 1000 in node 1 while all other nodes are initialized with value 0. Figure 3 shows how the value of each node asymptotically converges to the average of 250 for both protocols.

The *Push-Sum* as well as our *Disperser* protocol rely on a kind of *mass conservation*. In *Push-Sum*, half of a node's sum s is sent, the remainder is kept, but the overall sum remains constant. For *Disperser*, this conservation is obvious: The number of X molecules is conserved by all reactions. The two protocols differ in how they asymptotically approach the equilibrium: While *Push-Sum*'s code is executed isochronously, moving half of the value to a neighbor, *Disperser* transfers only one molecule

[3] Appendix B provides implementation details for the *Disperser* protocol.

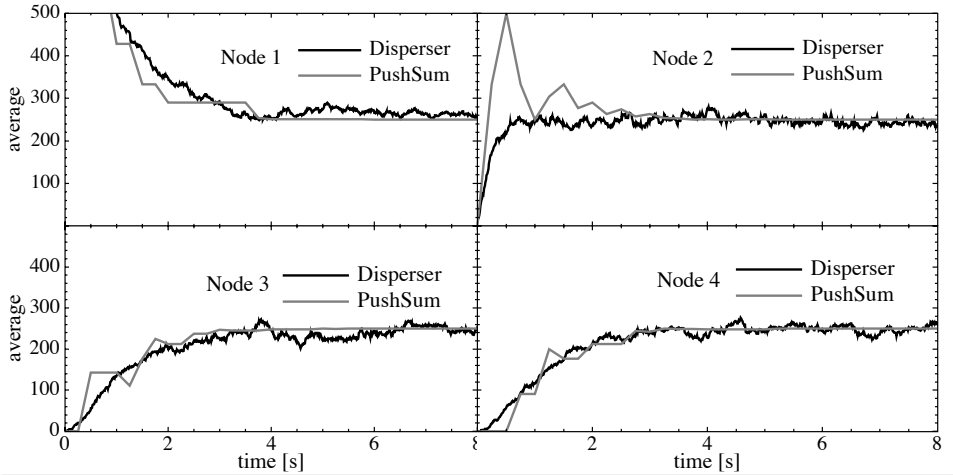

Figure 3: OMNeT++ simulation of the *PushSum* and the *Disperser* protocol.

per reaction, this rate being controlled by the inter-reaction time interval, which is inversely proportional to the concentration. The convergence time of both protocols can be lowered: In *PushSum* this is achieved by choosing a shorter update interval, whereas in *Disperser*, we have can speed up the reactions by increasing the number of C molecules.

While *PushSum* calculates the exact value of the average, *Disperser* only provides an approximation. On the other hand, the *PushSum* protocol shows a tendency to overshoot, even if the transmission links do not delay packets. This is not the case for *Disperser*, as can be seen in Fig. 3 (e.g. node 2).[4]

Note that neither of the two protocols is robust against the deletion of messages. However, while a lost message in *Push-Sum* results in the loss of half of a node's value, a packet loss in *Disperser* only decreases the value by one.

4 Fraglets — a Chemical Programming Language

So far we demonstrated static reaction networks where abstract reaction rules were "installed" permanently in each node. In this section, we extend this model aiming at dynamically changing the set of reaction rules. We present the Fraglets language [28, 1], an artificial chemistry [9], whose corresponding chemical machine is executable and which serves as a simple platform to run chemical protocols.

Each molecule $s \in \mathcal{S}$, or packet, is a string of symbols over a finite alphabet Σ. The first symbol of the string defines the string rewriting operation applied to this molecule by the virtual chemical machine: It can be thought of an assembler instruction. For example, the molecule `[fork a b c d]` transforms itself and splits into the two molecules `[a c d]` and `[b c d]`. The following list shows some essential instructions and their actions, whereas Appx. A lists and discusses additional Fraglets instructions:

$$
\begin{aligned}
[\texttt{match}\ \alpha\ \Phi] + [\alpha\ \Omega] &\longrightarrow [\Phi\ \Omega] \\
[\texttt{fork}\ \alpha\ \beta\ \Omega] &\longrightarrow [\alpha\ \Omega] + [\beta\ \Omega] \\
[\texttt{nop}\ \Omega] &\longrightarrow [\Omega] \\
[\texttt{send}\ k\ \Omega]_i &\longrightarrow [\Omega]_k \quad \text{if } (i,k) \in E
\end{aligned}
$$

$\alpha, \beta \in \Sigma$ are arbitrary symbols, $\Phi, \Omega \in \Sigma^*$ are symbol strings and $i, k \in V$ are network nodes. Molecules starting with **match** or any non-instruction identifier are in their *normal form*. The **match**

[4] This can even be formally verified by the fact that the eigenvalues of (8) only have real parts.

instruction can be used to join two molecules by concatenating the second to the first after removing the processed headers. Subsequent instructions immediately reduce the product further until they again reach their normal form. For example, the two molecules `[match pkt send 2 pkt]` and `[pkt data]` in node 1 imply the following reaction:

$$\begin{aligned} &\texttt{[match pkt send 2 pkt]}_1 + \texttt{[pkt data]}_1 \\ &\qquad\longrightarrow \texttt{[send 2 pkt data]}_1 \\ &\text{(transmission to neighbor 2)} \quad\longrightarrow \texttt{[pkt data]}_2 \end{aligned}$$

Such a chemical language allows us to "program" the reaction graph. Molecules now have a structure, i.e. they can *contain* information such as piggybacked user data. However, the dynamics of the reaction network is still governed by the law of mass action and thus, the protocol's behavior is chemically controlled.

5 Self-Healing Code

In order to heal itself, program code must be self-referential and capable of regenerating itself. In addition to these structural and behavioral properties, the code dynamics must be steered by a control loop that recognizes defects and triggers code self-repair. In this section we show how both, structural and dynamical aspects can be realized using chemical programs.

5.1 The "Quine"

In our chemical programming model presented in Sect. 4, there is no distinction between code and data. Molecules may contain code or data or both, and therefore it is possible to modify code or generate it on the fly. An example that illustrates this concept is the following "Quine"[5], a program that generates its own code as output [31]: `[match x fork nop match x]`, `[x fork nop match x]`. These two molecules react and, by doing so, according to the rewriting rules, regenerate themselves as shown in Fig. 4.

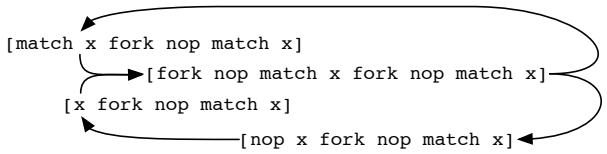

Figure 4: The chemical "Quine" is a set of molecules that replicates itself.

5.2 Code Homeostasis

The self-replicating quine above can easily be converted in one that generates two copies of itself in each round.[6] Due to the law of mass action, its concentration rises exponentially in time. To limit this unbounded growth, we apply a "non-selective dilution flow" to the reactor which randomly destroys a molecule whenever the total number of molecules exceeds a certain threshold. This imposes selective pressure to the reaction vessel; consequently, molecules that are not part of a self-replicating set will eventually be displaced.

[5] after the philosopher and logician Willard van Orman Quine (1908–2000) who studied indirect self-reference

[6] Duplicating quine: `[match x fork fork fork nop match x] + [x fork fork fork nop match x]`

Interestingly and crucially, this harsh environment, where several quines have to fight for resources, makes them tamper-proof: Even when destroying some of a quines' instances, the surviving population grows again until the vessel capacity is reached. In [21], we examined the robustness of the quines in detail. Note that the self-healing property does not require an external observer that would monitor the system and that would plan interventions: The overall system rather intrinsically controls itself to maintain a stable state, that is the *homeostasis* of program code.

6 A Path Load Balancing Protocol

We now make use of quines as building blocks in order to assemble a self-healing protocol that balances a packet stream over two different network paths such that packet loss is minimized.

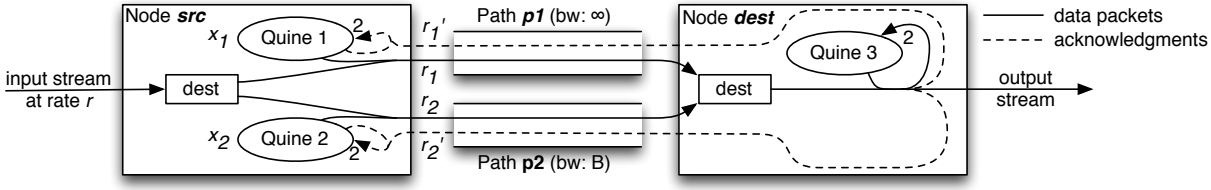

Figure 5: Data packets to node *dest* are injected at rate r into node *src*. Quines 1 and 2 (concentration x_1 and x_2, resp.) compete for and forward packets at rate r_1 and r_2 over paths *p1* and *p2*, resp. The quine's replication is controlled by acknowledgments received at rate r_1' and r_2', resp.

As depicted in Fig. 5, we inject packets at rate r into the source node where two quines, one for each path, compete for them and send them over the corresponding path. Instead of replicating as fast as possible, the quines wait for and react with acknowledgment packets. These acknowledgments are sent back over the reverse path by the third quine in the destination node that delivers the packet to the application. A Fraglets implementation of the protocol is provided and discussed in Appx. C.

Formal Convergence Proof

This scheme leads to a perfect packet balance among the two paths. When the source node's reaction vessel is saturated, its molecules either belong to quine 1 or 2, as other molecules have been squeezed out. Let's denote the relative concentrations by x_1 and x_2, respectively, satisfying $x_1 + x_2 = 1$. Since replication is triggered by received acknowledgments, these concentrations are

$$x_1 = \frac{r_1'}{r_1' + r_2'} \quad \text{and} \quad x_2 = \frac{r_2'}{r_1' + r_2'} \tag{10}$$

where r_n' is the rate of acknowledgments received over path pn.

Let's assume that the bandwidth of *p1* is infinite whereas *p2* drops packets exceeding a rate of B packets/s. We examine the overload situation where the total rate $r > 2B$. Consequently, the rate of acknowledgments is $r_1' = r_1$ and $r_2' = \min(r_2, B)$. Due to the law of mass action, the fraction of packets sent over *p1* is proportional to the concentration of quine 1:

$$r_1 = x_1 r = \frac{r_1}{r_1 + \min(r_2, B)} r \tag{11}$$

Hence,

$$r_1 = r - B \quad \text{and} \quad r_2 = r - r_1 = B \tag{12}$$

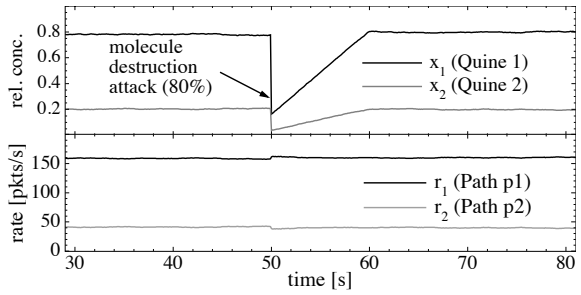

Figure 6: OMNeT++ simulation: Self-healing code in the *src* node during a deletion attack.

Quine 2 reduced its concentration so as to only forward packets up to the bandwidth limitation of path *p2*, as was to be proved.

6.1 Surviving Code Attacks

Like in Sect. 3, our load balancing CNP reaches a (chemical) equilibrium. Deviations from the equilibrium, for example by lost molecules on a network path, are compensated by increasing the population of quines that forward packets over the opposite path. Moreover, the code itself is organized in circuits of self-replicating molecules: If we forcefully destroy some code, the system will eventually regenerate it and, after some transition time, autonomously finds back to the equilibrium state. In fact, packet loss as well as code loss are treated by the same mechanism and in the same way.

Figure 6 depicts the protocol's response to such a deletion attack, simulated in OMNeT++ [2] for a packet injection rate of $r = 200\,\mathrm{pkts/s}$ and a bandwidth limitation for path *p2* of $B = 40\,\mathrm{pkts/s}$. At $t = 50\,\mathrm{s}$ we removed 80 % of all molecules (code and data) from the source vessel. Note that the code regenerates itself within 10 s while the traffic distribution r_1/r_2 remains unchanged.

6.2 Discussion

Being a showcase, our protocol is not as elaborate as existing load-aware protocols in many respects. Emerging from microscopic chemical reactions, it realizes an *intrinsic* bandwidth estimation by using the rate of actual data packets whereas most existing bandwidth estimation techniques numerically calculate the bandwidth based on the rate of separate probe packets or on their inter-arrival time (e.g. [11] for TCP).

Interestingly, our protocol converges for bursty traffic: The stochastic algorithm schedules the next reaction based on an exponential probability distribution; consequently, a burst of incoming packets appears blurred at the output, which helps estimating the bandwidth.

The load balancing protocol discussed in this paper may also be used as a building block in a larger context. In [22], it forms the self-adapting forwarding engine of a self-healing routing protocol implementation.

7 Relevance, Impact, Future Work

In this section we will discuss some of the consequences of a chemical protocol mindset and its derived solutions.

7.1 Determinism vs. Resilience

Although ODEs can be used to approximate the behavior of CNPs (as shown in Sects. 3 and 6) the underlying execution model is stochastic. Consequently, pure CNPs cannot be used for time-critical protocols. Additionally, CNPs cannot prevent packet reordering. However, the deterministic character of today's protocols is not needed everywhere in networking; there are network services where we can relax the level of certainty for the purpose of obtaining more resilience. This applies specifically to continuously running services like routing, load balancing and long lasting signaling streams. As we pointed out in Sect. 3, collective computations at large are also good candidates. The use of representation-free encoding mitigates the packet reordering problem since it does not matter which of the identical molecule instances is sent next.

Generally, we should try to soften the precision of the protocol's specification. As is conjectured in [27], this may avoid brittleness of the system and facilitates a CNP implementation. In return, we gain resilience to various perturbations, such as the partial loss of information in form of packets or code.

7.2 System Design and Analysis

Traditional flow-based protocols are analyzed as being composed of a fixed number of interconnected packet queues, scheduled by *deterministic* algorithms [19]. In CNPs we can identify similar entities, which however interact differently: Each node has a single packet buffer (the molecule multiset) and we impose a specific *stochastic* scheduling that is compatible with the law of mass action [14, 13]. We can regard the multiset as virtually divided into individual queues, one for each molecular type, that are interconnected by reaction rules. The rules may compete for the same type, akin to rule-based protocols [20, 10, 30], and form a (distributed) closed network in the sense of Kelly [17].

While traditionally only the packet flows have been modeled as stochastic processes, we also apply (and impose) this to the packet processing. On the analysis side, this does not change much: the same mathematical framework can be applied. However, our environment forces protocol designers to come up with "fluid" protocols that can continuously track the environment and adapt to it. This might seem to be a challenging task. But, in addition to a sound mathematical foundation, a CNP approach potentially promotes good system-wide properties when composing such protocols. CNPs also provide a graphical notation, the reaction graph, which is intuitively graspable, and from which the underlying mathematical equations can be derived automatically. We conjecture that this may simplify protocol and system design.

7.3 Resource Exploitation vs. Embodiment

CNPs tend to exploit all available resources, leading to a competitive rather than a cooperative environment. A contribution to last year's workshop promoted this strategy as long as the marginal cost can be driven lower than the marginal benefit [26]. This tradeoff, however, has to be identified and analyzed for each problem separately.

While exploiting resources, CNPs at the same time explore and dynamically adapt to the environment. Our load balancing protocol is able to obtain bandwidth information *because* it fills the link with packets; the quines occupy the available memory but survive when reducing it. Hence, CNPs achieve a high degree of embodiment [25]: computation may be outsourced to the environment, a concept already employed in robotics [24].

7.4 Towards Self-Optimizing Protocols

In Computer Science, bio-inspired approaches recently gained attention mostly because of the promise of robustness and scalability. The ultimate property however, evolvability, is a challenge due to the lack of a formal theory and the non-determinism of the resulting solutions. With a chemical protocol design, a gradual path from static to self-healing, and even towards self-optimizing and evolving protocols, could be envisioned. Mutations could be useful to evolve protocols online, potentially enabling self-optimization and long-term adaptability. We conjecture that CNPs are better suited to automatic evolution than traditional protocols due to their inherent property of multi-stability that lets them glide into the next equilibrium according to environmental conditions.

8 Conclusions

In this paper we looked at networking protocols with the eye of a chemist. We examined how chemical concepts can be transposed to network design in order to obtain the same emergent properties that we find in chemical systems such as stable equilibria, nota bene gaining analyzability. This requires a paradigmatic shift from designing local state machines to weaving global reaction networks. For the first time, we demonstrated how to obtain a protocol that intrinsically heals itself from code deletion attacks. We believe that such techniques open the door to the design of truly robust networks which are able to survive even in environments where reliable code execution can not be taken for granted.

Acknowledgments

This work has been supported by the Swiss National Science Foundation through SNF Project Self-Healing Protocols (2000201-109563). We would also like to thank Lidia Yamamoto for her continuous contribution to our research, and Christophe Jelger and Pierre Imai for their useful review comments.

A The Fraglets Programming Language

Fraglets is a programming language and execution model for chemically inspired programs, especially targeted for networking protocol design [28]. In this section, we briefly introduce the Fraglets interpreter and list some essential instructions. Note however, that because Fraglets is currently used for research projects, the instruction set as well as the implementation details may change from time to time.

A.1 Interpreter

There is a reference implementation of the Fraglets interpreter [1]. The interpreter reads and parses an input file, containing a description of the initial set of molecules, and simulates the reactions according to the instruction set and the law of mass action. The following example shows a simple input file:

```
# File abba.fra: Simple A <-> B reaction
f [matchp x y]    # Convert [x] to [y]
f [matchp y x]    # Convert [y] to [x]
# Initial concentration of [x]
f [x]1000         # Start with 1000 [x] molecules
```

Note that you have to prepend the command `f` to inject a fraglet into the reaction vessel. When placing an integer after a fraglet (e.g. `[x]1000`), 1000 instances of that fraglet are injected. The Fraglets program above is simulated by executing the command:

```
fraglets < abba.fra
```

For further details on how to install, configure and run Fraglets programs, please refer to [1].

A.2 Instruction Set

The following two tables show an excerpt of the Fraglets instruction set. A molecule $s \in \Sigma^*$ is a string over an alphabet Σ. The first symbol of a molecule is always treated as instruction identifier, specifying how the molecule is rewritten. We use the variables $\alpha, \beta, \chi \in \Sigma$ for arbitrary symbols, $\Omega, \Phi, \Psi \in \Sigma^*$ for symbol strings, and $i, k \in V$ for node identifiers.

The **match** and **matchp** instructions induce a bimolecular reaction between the molecule starting with **match** or **matchp** and a molecule starting with the same symbol as the first molecule's second symbol (as shown in Tab. 1).

Reactants $\longrightarrow$ **Products**	**Description**
[**match** α Φ] + [α Ω] $\longrightarrow$ [Φ Ω]	Joins two molecules by concatenating their tails
[**matchp** α Φ] + [α Ω] $\longrightarrow$ [**matchp** α Φ] + [Φ Ω]	Persistent version of **match**

Table 1: Fraglets instructions for programming bimolecular reactions.

For example, the molecule `[matchp x y]` rewrites the header symbol of another molecule from `x` to `y`:

$$\texttt{[matchp x y]} + \texttt{[x some data]} \longrightarrow \texttt{[matchp x y]} + \texttt{[y some data]} \tag{13}$$

If a molecule starts with a transformation instruction, such as the ones listed in Tab. 2, the instruction is immediately executed as well as all subsequent transformations of the resulting products.

Reactants $\longrightarrow$ Products	Description
$[\texttt{nop}\ \Omega] \longrightarrow [\Omega]$	No operation
$[\texttt{fork}\ \alpha\ \beta\ \Omega] \longrightarrow [\alpha\ \Omega] + [\beta\ \Omega]$	Splits into two molecules, having different header symbols
$[\texttt{split}\ \Omega\ \star\ \Phi] \longrightarrow [\Omega] + [\Phi]$	Splits into two molecules at the first occurrence of symbol $\star$
$[\texttt{splitat}\ \alpha\ \Omega\ \alpha\ \Phi] \longrightarrow [\Omega] + [\Phi]$	Splits into two molecules at the first occurrence of symbol α
$[\texttt{releaseat}\ \alpha\ \Omega\ \alpha\ \Phi\ \alpha\ \Psi] \longrightarrow [\Omega\ \Psi] + [\Phi]$	Releases an interior part, enclosed by symbols α, as a separate molecule
$[\texttt{deliver}\ \Omega]_i \longrightarrow [\Omega]_{i\text{'s application}}$	Node i sends the tail to the application
$[\texttt{send}\ k\ \Omega]_i \longrightarrow \begin{cases} [\Omega]_k & \text{if } (i,k) \in E, \\ \emptyset & \text{otherwise.} \end{cases}$	Node i sends the tail to neighbor k
$[\texttt{send all}\ \Omega]_i \longrightarrow [\Omega]_k \quad \forall (i,k) \in E$	Node i broadcasts the tail to all neighbors

Table 2: Fraglets transformations that are immediately reduced

For example, the following fraglet is immediately reduced until its normal form is reached. A fraglet is in *normal form* if it either starts with a bimolecular reaction instruction like **`match`** or **`matchp`**, or with any identifier symbol that is not a transformation instruction, such as `x`, `data`, etc.

$$\begin{aligned} &\texttt{[nop nop split match x y * x data]} \\ &\longrightarrow \texttt{[nop split match x y * x data]} \\ &\longrightarrow \texttt{[split match x y * x data]} \\ &\longrightarrow \texttt{[match x y]} + \texttt{[x data]} \end{aligned} \tag{14}$$

Note that transformations always reduce the size of the molecule by at least one in each rewriting step. This guarantees that there are no infinite transformation loops. The only way of elongating a molecule is to concatenate it with another one by using **`match`** and **`matchp`** instructions. Also note that mass (number of symbols) is not conserved: For example, the **`nop`** instruction destroys a symbol while the **`fork`** instruction almost doubles the number of symbols from reactant to products.

B *Disperser* Protocol Details

This section contains supplementary information to the chemically inspired *Disperser* protocol, introduced in Sect. 3.2. We show how this protocol can be "programmed" using Fraglets – an executable chemistry.

B.1 Special Case: *Disperser* Implementation for Known Topologies

Consider the simple four node topology depicted in Fig. 7. If, like in this case, the topology is known, we can install a simple Fraglets program into each node that realizes the *Disperser* reaction

$$C_{ik} + X_i \xrightarrow{r_i} C_{ik} + X_k \qquad \forall (i,k) \in E \tag{15}$$

Molecule X may be represented by fraglet `[x]`. For node *n1* the pre-installed program realizing (15) for the topology shown in Fig. 7 would then be

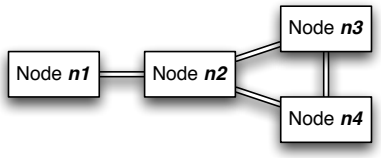

Figure 7: Example topology for the *Disperser* protocol.

```
# Disperser program for node n1
f [matchp x send n2 x]    # C_12: Send [x] to neighbor n2
# Initialize the local value
f [x]1000                 # X_1: Inject 1000 x molecules
```

This persistent rule reacts with an `x` molecule and sends it to neighbor *n2*:

$$\texttt{[matchp x send n2 x]}_{n1} + \texttt{[x]}_{n1} \longrightarrow \texttt{[matchp x send n2 x]}_{n1} + \underbrace{\texttt{[send n2 x]}_{n1}}_{\xrightarrow{\text{sent}} \texttt{[x]}_{n2}} \quad (16)$$

Similarly, the program for node *n2* reads

```
# Disperser program for node n2
f [matchp x send n1 x]    # C_21: Send [x] to neighbor n1
f [matchp x send n3 x]    # C_23: Send [x] to neighbor n3
f [matchp x send n4 x]    # C_24: Send [x] to neighbor n4
```

Node *n3* contains the following program:

```
# Disperser program for node n3
f [matchp x send n2 x]    # C_32: Send [x] to neighbor n2
f [matchp x send n4 x]    # C_34: Send [x] to neighbor n4
```

and accordingly, the program for node *n4* is

```
# Disperser program for node n4
f [matchp x send n2 x]    # C_42: Send [x] to neighbor n2
f [matchp x send n3 x]    # C_43: Send [x] to neighbor n3
```

B.2 General Case: *Disperser* Implementation for Unknown Topologies

If a node does not know the identifiers of its neighbor nodes, we cannot use pre-configured reaction rules. In this case we may use the `send all` instruction to broadcast an active molecule to all neighbors. There the molecule reads the local node identifier and sends a unicast message back to the originating node. This learned information can then be used to generate the C_{ik} molecules on the fly. We presented this method in [23].

C Load Balancing Protocol Details

In Sect. 6 we introduced a self-healing path load balancing protocol. Here, we present its Fraglets code and explain the protocol's operation in more detail. This section is organized as follows: In Sect. C.1, we first present the setup for our experiments together with the initial program code for all participating nodes. Then, in Sect. C.2, we present a generic recipe of how to generate this code for arbitrary data transfer paths. Finally, Sect. C.3 provides a detailed description of the reaction networks spanned by the code, by following the journey of a data packet from source to destination.

C.1 Experimental Setup and Program Code

The experiment in Sect. 6 was carried out in OMNeT++ [2] using a simple four node topology as depicted in Fig. 8. There are two intermediate nodes between the source and the destination node, allowing for two different paths (*p1* and *p2*).

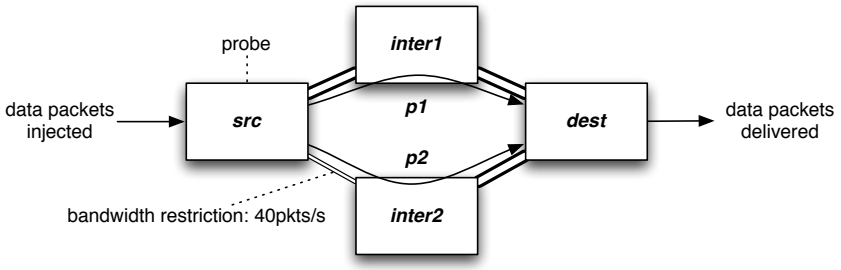

Figure 8: Topology for the experiment carried out in Sect. 6.

In the following, we present the molecules injected into the four nodes at the start of the experiment. A detailed program walkthrough is provided later in Sect. C.3.

C.1.1 Program Code for the Source Node

The source node (*src*) is initialized with the following Fraglets program consisting of the bootstrapping code for two different forwarding quines, one for each path.

```
# Randomly destroy molecules if the vessel contains more than 5000
o dilute: fraglet-capacity=5000

# Forwarding quine to 'dest' via 'inter1' (path p1)
f [fork nop BP1 match dest
   send inter1 dest
   send src
   splitat _ACK match BP1 fork fork fork nop BP1 _ACK]

# Forwarding quine to 'dest' via 'inter2' (path p2)
f [fork nop BP2 match dest
   send inter2 dest
   send src
   splitat _ACK match BP2 fork fork fork nop BP2 _ACK]
```

C.1.2 Program Code for the Intermediate Nodes

The source code of the two intermediate nodes each consist of a single forwarding quine, similar to the ones in the source node. For intermediate node 1 (*inter1*) the code is as follows:

```
# Randomly destroy molecules if the vessel contains more than 5000
o dilute: fraglet-capacity=5000

# Forwarding quine to 'dest' via 'dest' (path p1)
f [fork nop BP1 match dest
   send dest dest
   send inter1
   splitat _ACK match BP1 fork fork fork nop BP1 _ACK]
```

The source code for intermediate node 2 (*inter2*) is similar:

```
# Randomly destroy molecules if the vessel contains more than 5000
o dilute: fraglet-capacity=5000

# Forwarding quine to 'dest' via 'dest' (path p1)
f [fork nop BP1 match dest
   send dest dest
   send inter2
   splitat _ACK match BP1 fork fork fork nop BP1 _ACK]
```

C.1.3 Program Code for the Destination Node

The destination node needs a slightly different molecule in order to deliver the data packets to the application. It's source code is as follows:

```
# Randomly destroy molecules if the vessel contains more than 5000
o dilute: fraglet-capacity=5000

# Delivery quine (any path)
f [fork nop BP1 match dest
   releaseat _DATA split match BP1 fork fork fork nop BP1 *
   deliver _DATA]
```

C.2 Program Recipe

Attentive readers may have noticed that the initial Fraglets code for the source and the intermediate nodes is similar. In this section we present a recipe of how to generate program code for each node of a transmission path.

Networking protocols always allow for a global and a local point of view. Considered globally, the protocol controls packet transport paths from source to destination over multiple intermediate hops as depicted in Fig. 9. Such a transport path is formally described by an ordered set $P = (i_0, i_1, \ldots, i_N, d)$ consisting of the source node $i_0 \in V$, several intermediate nodes $i_1, i_2, \ldots, i_N \in V$, and the destination node $d \in V$. Let's further denote the ordered set of nodes along path P without the destination node as $P' = (i_0, i_1, \ldots, i_N)$.

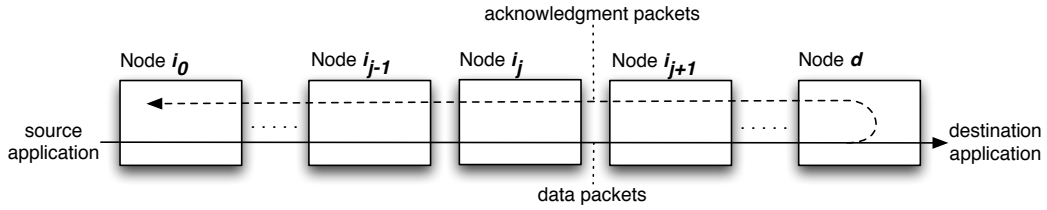

Figure 9: A transport path leads from a source node (i_0) over multiple intermediate nodes ($i_1, \ldots, i_N$) to the destination node (d). Data packets are sent downstream from source to destination while acknowledgment packets are send back upstream, if the data packets arrive at the destination.

C.2.1 Forwarding Quine Template

Considered locally, each node $i_j \in P'$ along the path (exclusive of the destination node d) contains a quine that forwards a packet to the destination d via the next hop i_{j+1}, waits for the acknowledgment and replicates in turn. The *forwarding quine* is described by the following template molecule, where the parameters $x, y, z \in V$ denote the local node (x), the next hop along the path (y) and the destination node (z):

$$[\texttt{fork nop } b_{x,y,z} \texttt{ match } z \texttt{ send } y\ d\ ack_{x,y,z}]_x \tag{17}$$

where

$$ack_{x,y,z} := \texttt{send } x \texttt{ splitat \_ACK } duplicate_{x,y,z} \texttt{ \_ACK} \tag{18}$$

$$duplicate_{x,y,z} := \texttt{match } b_{x,y,z} \texttt{ fork fork fork nop } b_{x,y,z} \tag{19}$$

For each forwarding quine, we need a node-unique identifier $b_{x,y,z}$, that serves as its blueprint identification tag, needed for replication.

C.2.2 Delivery Quine Template

The destination node of each path $d \in P$ contains a special quine, the *delivery quine*. Unlike the forwarding quine, the delivery quine replicates immediately, sends the acknowledgments upstream and delivers the packet's data part to the application. Again, the delivery quine can be described as a template molecule where we only have a single template parameter $z \in V$, representing the local (as well as the destination) node:

$$[\texttt{fork nop } b_z \texttt{ match } z \texttt{ releaseat \_DATA split } duplicate_z \texttt{ * deliver \_DATA}]_z \tag{20}$$

where

$$duplicate_z := \texttt{match } b_z \texttt{ fork fork fork nop } b_z \tag{21}$$

Again, for each delivery quine, we need a node-unique identifier b_z.

C.3 Protocol Walk-through

In the following, we provide a detailed description of the reactions that are executed when forwarding a packet from source to destination. We first focus on how a packet is forwarded to the destination. Then

we cover the reactions responsible for data delivery and creation of an acknowledgment packet in the destination node. Finally, we show how the acknowledgment packet travels upstream to the source node, replicating all forwarding quines in turn.

C.3.1 Downstream Forwarding of Data Packets

Let's examine the forwarding quine's reaction trace along a path $P = (i_0, i_1, \ldots, i_N, d)$. The reaction network of the forwarding quine in one of the intermediate nodes is depicted in Fig. 10.

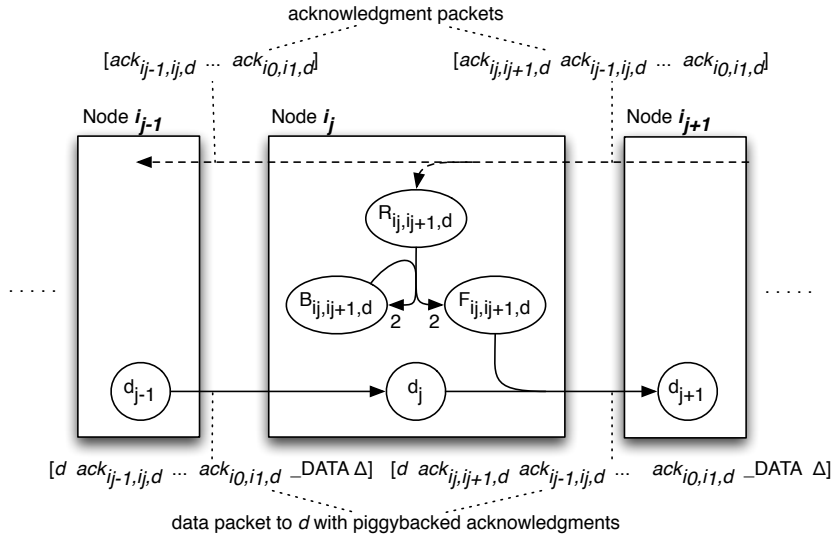

Figure 10: Reaction network of the forwarding quine.

In each node $i_j \in P'$, the first **`fork`** instruction of the instantiated template (17) is immediately executed, yielding the blueprint $B_{i_j,i_{j+1},d}$ and an identical, active version, $F_{i_j,i_{j+1},d}$, that forwards a packet destined to d via the next hop i_{j+1}:

$$[\texttt{fork nop } b_{i_j,i_{j+1},d} \texttt{ match } d \texttt{ send } i_{j+1}\ d\ ack_{i_j,i_{j+1},d}]_{i_j}$$
$$\longrightarrow \overbrace{[\texttt{match } d \texttt{ send } i_{j+1}\ d\ ack_{i_j,i_{j+1},d}]_{i_j}}^{F_{i_j,i_{j+1},d}} + \overbrace{[b_{i_j,i_{j+1},d} \texttt{ match } d \texttt{ send } i_{j+1}\ d\ ack_{i_j,i_{j+1},d}]_{i_j}}^{B_{i_j,i_{j+1},d}} \quad (22)$$

The journey of a data packet destined to d starts when it is injected into the source node i_0. It has the following format:

$$\overbrace{[d \texttt{ \_DATA } \Delta]_{i_0}}^{d_{i_0}} \quad (23)$$

where the first symbol is the identifier of the destination node (d), the second symbol is the constant identifier `_DATA` that separates the header from the user data, and the tail $\Delta \in \Sigma^*$ is the actual data string. This packet reacts with the forwarding quine's active rule ($F_{i_0,i_1,d}$), which forwards the data packet to the corresponding next hop node i_1:

$$\overbrace{[\texttt{match } d \texttt{ send } i_1\ d\ ack_{i_0,i_1,d}]_{i_0}}^{F_{i_0,i_1,d}} + \overbrace{[d \texttt{ \_DATA } \Delta]_{i_0}}^{d_{i_0}} \xrightarrow{\text{sent}} \overbrace{[d\ ack_{i_0,i_1,d} \texttt{ \_DATA } \Delta]_{i_1}}^{d_{i_1}} \quad (24)$$

Note that the local acknowledgment $ack_{s,i_1,d}$, which is needed to replicate the quine, is prepended to the data packet akin to a reverse path in Ad Hoc routing protocols. This process is repeated by every node $i_j \in P'$, such that when arriving at destination node d, the data packet looks as follows:

$$\overbrace{[d\ ack_{i_N,d,d}\ \ldots\ ack_{i_2,i_3,d}\ ack_{i_1,i_2,d}\ ack_{i_0,i_1,d}\ \texttt{\_DATA}\ \Delta]_d}^{d_d} \tag{25}$$

C.3.2 Data Delivery

In the destination node d, the protocol has to extract the actual data Δ from the packet, deliver it to the application and send the "reverse path acknowledgments" back. This is done by the delivery quine. Its reaction network is depicted in Fig. 11.

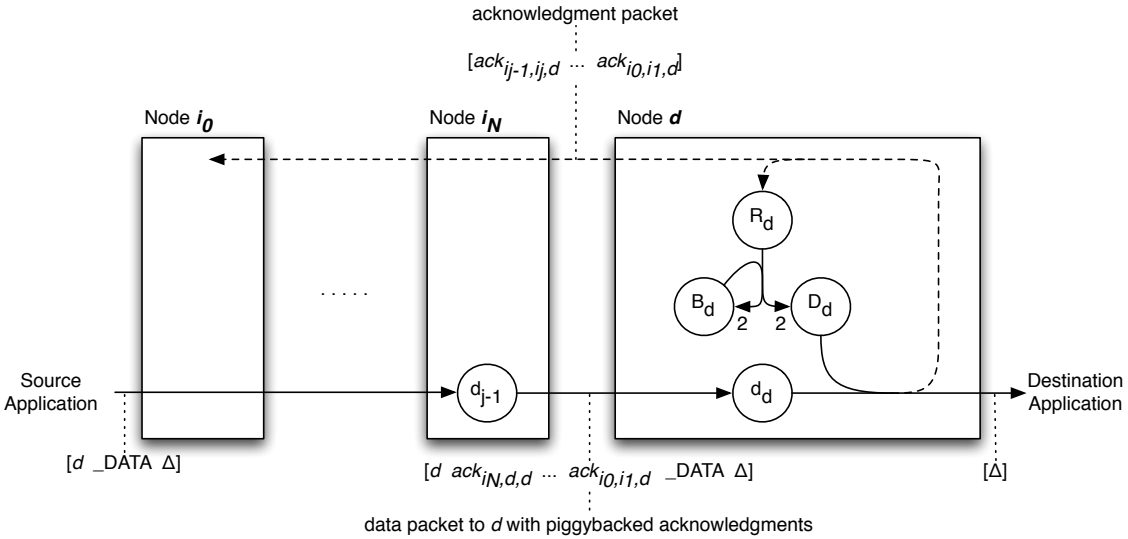

Figure 11: Reaction network of the delivery quine.

The first **`fork`** instruction of the instantiated template (20) is immediately executed, yielding the blueprint B_d and an identical, active version, D_d, that delivers a packet and sends back an acknowledgment:

$$\begin{aligned}&[\texttt{fork nop}\ b_d\ \texttt{match}\ d\ \texttt{releaseat \_DATA split}\ duplicate_d\ \star\ \texttt{deliver \_DATA}]_d \\ &\longrightarrow \overbrace{[\texttt{match}\ d\ \texttt{releaseat \_DATA split}\ duplicate_d\ \star\ \texttt{deliver \_DATA}]_d}^{D_d} \\ &+ \underbrace{[\ b_d\ \texttt{match}\ d\ \texttt{releaseat \_DATA split}\ duplicate_d\ \star\ \texttt{deliver \_DATA}]_d}_{B_d}\end{aligned} \tag{26}$$

When D_d reacts with a received packet d_d, the resulting product is immediately split into three different molecules: (i) a data molecule that is delivered to the application, (ii) an acknowledgment packet that is sent back over the same path to trigger replication of the forwarding quines, and (iii) a reward molecule (R_d) that triggers replication of the local delivery quine:

$$\overbrace{\texttt{[match } d \texttt{ releaseat \_DATA split } duplicate_d \texttt{ * deliver \_DATA]}_d}^{D_d}$$
$$+ \underbrace{\texttt{[}d\ ack_{i_N,d,d}\ \ldots\ ack_{i_0,i_1,d}\ \texttt{\_DATA}\ \Delta\texttt{]}_d}_{d_d}$$
$$\longrightarrow \texttt{[releaseat \_DATA split } duplicate_d \texttt{ * deliver \_DATA}$$
$$ack_{i_N,d,d}\ \ldots\ ack_{i_0,i_1,d}\ \texttt{\_DATA}\ \Delta\texttt{]}_d \tag{27}$$
$$\longrightarrow \texttt{[split } duplicate_d \texttt{ * deliver } \Delta\texttt{]}_d + \texttt{[}ack_{i_N,d,d}\ \ldots\ ack_{i_0,i_1,d}\texttt{]}_d \tag{28}$$
$$\longrightarrow \underbrace{\texttt{[}duplicate_d\texttt{]}_d}_{R_d} + \texttt{[deliver } \Delta\texttt{]}_d + \texttt{[}ack_{i_N,d,d}\ \ldots\ ack_{i_0,i_1,d}\texttt{]}_d \tag{29}$$

The molecule `[deliver Δ]` delivers the actual data (Δ) to the application. The string of acknowledgments can be rewritten using (18) and transformed as follows:

$$\texttt{[}ack_{i_N,d,d}\ \ldots\ ack_{i_0,i_1,d}\texttt{]}_d$$
$$\overset{(18)}{\equiv} \texttt{[send } i_N \texttt{ splitat \_ACK } duplicate_{i_N,d,d} \texttt{ \_ACK } ack_{i_{N-1},i_N,d}\ \ldots\ ack_{i_0,i_1,d}\texttt{]}_d$$
$$\xrightarrow{\text{sent}} \texttt{[splitat \_ACK } duplicate_{i_N,d,d} \texttt{ \_ACK } ack_{i_{N-1},i_N,d}\ \ldots\ ack_{i_0,i_1,d}\ \texttt{]}_{i_N} \tag{30}$$

The chain of acknowledgments is sent back to the last intermediate node i_N. This is possible because every ack_x, y, z first sends itself from the downstream neighbor y to its origin, x. There, the acknowledgment is processed as discussed in the following section.

C.3.3 Upstream Forwarding of Acknowledgment Packets

In the previous section we noticed that the destination node sends back the piggybacked acknowledgments to the last intermediate node of the path, i_N. There, it is immediately executed and splits off the local reward:

$$\texttt{[splitat \_ACK } duplicate_{i_N,d,d} \texttt{ \_ACK } ack_{i_{N-1},i_N,d}\ \ldots\ ack_{i_0,i_1,d}\texttt{]}_{i_N}$$
$$\longrightarrow \overbrace{\texttt{[}duplicate_{i_N,d,d}\texttt{]}_{i_N}}^{R_{i_N,d,d}} + \texttt{[}ack_{i_{N-1},i_N,d}\ \ldots\ ack_{i_0,i_1,d}\texttt{]}_{i_N} \tag{31}$$

The first molecule (when expanded according to (19)) is considered as the reward for the local forwarding quine:

$$\overbrace{\texttt{[}duplicate_{i_N,d,d}\texttt{]}_{i_N}}^{R_{i_N,d,d}} \overset{(19)}{\equiv} \overbrace{\texttt{[match } b_{i_N,d,d} \texttt{ fork fork fork nop } b_{N,d,d} \texttt{]}_{i_N}}^{R_{i_N,d,d}} \tag{32}$$

This reward reacts with the local forwarding quine's blueprint ($B_{i_N,d,d}$) and, by doing so, duplicates the blueprint and the active forwarding rule ($F_{i_N,d,d}$):

$$
\begin{aligned}
&\overbrace{\texttt{[match } b_{i_N,d,d} \texttt{ fork fork fork nop } b_{i_N,d,d} \texttt{]}_{i_N}}^{R_{i_N,d,d}} \\
&\quad + \overbrace{\texttt{[}b_{i_N,d,d} \texttt{ match } d \texttt{ send } d\ d\ ack_{i_N,d,d}\texttt{]}_{i_N}}^{B_{i_N,d,d}} \\
&\longrightarrow \texttt{[fork fork fork nop } b_{i_N,d,d} \texttt{ match } d \texttt{ send } d\ d\ ack_{i_N,d,d}\texttt{]}_{i_N} && (33)\\
&\longrightarrow 2\texttt{[fork nop } b_{i_N,d,d} \texttt{ match } d \texttt{ send } d\ d\ ack_{i_N,d,d}\texttt{]}_{i_N} && (34)\\
&\longrightarrow 2\underbrace{\texttt{[match } d \texttt{ send } d\ d\ ack_{i_N,d,d}\texttt{]}_{i_N}}_{F_{i_N,d,d}} + 2\underbrace{\texttt{[} b_{i_N,d,d} \texttt{ match } d \texttt{ send } d\ d\ ack_{i_N,d,d}\texttt{]}_{i_N}}_{B_{i_N,d,d}} && (35)
\end{aligned}
$$

Note that this reaction a truly self-replicating, since by executing (and reducing) itself, the acknowledgment generates the forwarding rule that will piggyback the same acknowledgment to the next data packet. This is only possible because of the information stored in the blueprint, which is also regenerated by this reaction.

The remainder of the received acknowledgment packet, contains the acknowledgments of the other upstream nodes. Expanded according to (18), it is immediately executed as follows:

$$
\begin{aligned}
&\texttt{[}ack_{i_{N-1},i_N,d}\ \ldots\ ack_{i_0,i_1,d}\texttt{]}_{i_N} \\
&\overset{(18)}{\equiv} \texttt{[send } i_{N-1} \texttt{ splitat \_ACK } duplicate_{i_{N-1},i_N,d} \texttt{ \_ACK } ack_{i_{N-2},i_{N-1},d}\ \ldots\ ack_{i_0,i_1,d}\texttt{]}_{i_N} \\
&\quad \xrightarrow{\text{sent}} \texttt{[splitat \_ACK } duplicate_{i_{N-1},i_N,d} \texttt{ \_ACK } ack_{i_{N-2},i_{N-1},d}\ \ldots\ ack_{i_0,i_1,d}\texttt{]}_{i_{N-1}} && (36)
\end{aligned}
$$

In the upstream node i_{N-1}, this molecule is again processed according to (31), replicating that node's quine, and sending the remaining acknowledgments further upstream until the source node is reached.

References

[1] Fraglets Home Page. `http://www.fraglets.net`.

[2] OMNeT++ Community Site. `http://www.omnetpp.org`.

[3] H.I. Abrash. Studies Concerning Affinity. *J. Chem. Ed.*, 63:1044–1047, 1986. English translation of [29].

[4] R. Bajcsy and J. Košecká. The Problem of Signal and Symbol Integration. In *Advances in Artificial Intelligence*, volume 981 of *LNCS*, 1995.

[5] L.N. Chakrapani, P. Korkmaz, B.E.S. Akgul, and K.V. Palem. Probabilistic System-on-a-Chip Architectures. *ACM Trans. Des. Autom. Electron. Syst.*, 12(3):1–28, 2007.

[6] P. Dayan and L.F. Abbott. *Theoretical Neuroscience*. MIT Press, 2001.

[7] G. Di Caro and M. Dorigo. AntNet: Distributed Stigmergetic Control for Communications Networks. *J. Art. Intel. Res.*, 9:317–365, 1998.

[8] P. Dittrich and P. Speroni di Fenizio. Chemical Organization Theory. *Bull. Math. Bio.*, 69(4):1199–1231, 2007.

[9] P. Dittrich, J. Ziegler, and W. Banzhaf. Artificial Chemistries - A Review. *Artificial Life*, 7(3):225–275, 2001.

[10] F. Dressler, I. Dietrich, R. German, and B. Krüger. A Rule-Based System for Programming Self-Organized Sensor and Actor Networks. *Comput. Netw.*, 53(10):1737–1750, 2009.

[11] Y. Easwaran and M.A. Labrador. Evaluation and Application of Available Bandwidth Estimation Techniques to Improve TCP Performance. In *Proc. 29th Annual IEEE Int. Conf. Local Comp. Net.*, pages 268–275, 2004.

[12] P.T. Eugster, R. Guerraoui, A.-M. Kermarrec, and L. Massoulié. From Epidemics to Distributed Computing. *IEEE Computer*, 37(5):60–67, 2004.

[13] M.A. Gibson and J. Bruck. Efficient Exact Stochastic Simulation of Chemical Systems with Many Species and Many Channels. *J. Phys. Chem. A*, 104(9):1876–1889, 2000.

[14] D.T. Gillespie. Exact Stochastic Simulation of Coupled Chemical Reactions. *J. Phys. Chem.*, 81(25):2340–2361, 1977.

[15] J.H.S. Hofmeyr. Metabolic Control Analysis in a Nutshell. In *Proc. 2nd Int. Conf. Sys. Bio.*, pages 291–300, Madison, WI, USA, 2001.

[16] V. Jacobson. Congestion Avoidance and Control. In *Proc. Symp. Comm. Arch. Prot.*, pages 314–329, New York, NY, USA, 1988.

[17] F.P. Kelly. *Reversibility and Stochastic Networks*. John Wiley and Sons Ltd., 1979.

[18] D. Kempe, A. Dobra, and J. Gehrke. Gossip-based Computation of Aggregate Information. In *Proc. 44th IEEE Symp. Found. Comp. Sc.*, pages 482–491, 2003.

[19] J.-Y. Le Boudec and P. Thiran. *Network Calculus*, volume 2050 of *LNCS*. Springer, 2004.

[20] L.F. Mackert and I.B. Neumeier-Mackert. Communicating Rule Systems. In *Proc. IFIP WG6.1 7th Int. Conf. Prot. Spec., Test. and Verific.*, pages 77–88, 1987.

[21] T. Meyer, D. Schreckling, C. Tschudin, and L. Yamamoto. Robustness to Code and Data Deletion in Autocatalytic Quines. In *Transactions on Computational Systems Biology X*, volume 5410 of *LNBI*, pages 20–40. Springer, 2008.

[22] T. Meyer, L. Yamamoto, and C. Tschudin. A Self-Healing Multipath Routing Protocol. In *Proc. 3rd Int. Conf. on Bio-Insp. Models of Netw., Inform., and Comp. Sys. (BIONETICS 2008)*, 2008.

[23] T. Meyer, L. Yamamoto, and C. Tschudin. An Artificial Chemistry for Networking. In *Bio-Inspired Computing and Communication*, volume 5151 of *LNCS*, pages 45–57. 2008.

[24] R. Pfeifer, M. Lungarella, and F. Iida. Self-Organization, Embodiment, and Biologically Inspired Robotics. *Science*, 317(5853):1088–1093, 2007.

[25] T. Quick, K. Dautenhahn, C.L. Nehaniv, and G. Roberts. The Essence of Embodiment: A Framework for Understanding and Exploiting Structural Coupling Between System and Environment. In *Proc. 3rd Int. Conf. on Computing Anticipatory Systems (CASYS'99)*, volume 517, pages 649–660, 2000.

[26] R. Raghavendra, R. Mahajan, J. Padhye, and B. Zill. Eat All You Can in an All-you-can-eat Buffet: A Case for Aggressive Resource Usage. In *Proc. 7th ACM Workshop on Hot Topics in Networks (Hotnets VII)*, 2008.

[27] M. Shaw. "Self-healing": Softening Precision to Avoid Brittleness. In *Proc. 1st Workshop on Self-healing Systems*, pages 111–114, 2002.

[28] C. Tschudin. Fraglets - a Metabolistic Execution Model for Communication Protocols. In *Proc. 2nd Symp. on Aut. Intel. Net. and Sys. (AINS)*, 2003.

[29] P. Waage and C.M. Guldberg. Studies Concerning Affinity. *Forhandlinger: Videnskabs - Selskabet i Christiania*, 35, 1864.

[30] T. Weise, M. Zapf, and K. Geihs. Rule-based Genetic Programming. In *Proc. 2nd Int. Conf. on Bio-Insp. Models of Netw., Inform., and Comp. Sys. (BIONETICS 2007)*, pages 8–15, 2007.

[31] L. Yamamoto, D. Schreckling, and T. Meyer. Self-Replicating and Self-Modifying Programs in Fraglets. In *Proc. 2nd Int. Conf. on Bio-Insp. Models of Netw., Inform., and Comp. Sys. (BIONETICS 2007)*, 2007.