Experiments on the Automatic Evolution of Protocols Using Genetic Programming

Lidia Yamamoto and Christian Tschudin

Computer Science Department, University of Basel,
Bernoullistrasse 16, CH-4056 Basel, Switzerland
Lidia.Yamamoto@unibas.ch, Christian.Tschudin@unibas.ch

Abstract. Truly autonomic networks ultimately require *self-modifying, evolving* protocol software. Otherwise humans must intervene in every situation that has not been anticipated at design time. For this to become feasible autonomic systems must ensure non-disruptive on-line software evolution. We investigate related code steering techniques in two directions: One is the fully automatic selection of protocol service elements where, depending on device characteristics and current operation environment, each communication entity has to select among a potentially wide variety of protocol implementations providing similar services. The other direction relates to the automatic synthesis of new protocol elements which are the result of optimizing existing implementations for a specific context. In both cases we look at genetic programming as a tool to generate new code and software configurations automatically. In this paper we propose a framework for such a resilient protocol evolution and report on first exploratory results on the adaptation and re-adaptation to environmental conditions, and the elimination of superfluous code.

Keywords: protocol synthesis, protocol evolution, genetic programming.

1 Introduction

Managing change in a network and its services is currently a labor intensive task which is not automated. Any new algorithm must be engineered, then programmed, and deployed in the network. Today this process is slow and requires the effort of many people (network managers, engineers, programmers), which is outside the scope of autonomic networks. Networking software must be able to adapt and reconfigure – i.e., to evolve – by itself in the most autonomous way possible.

Ultimately, protocols and algorithms for autonomic networks should evolve during their own execution, with minimum service disruption. Such long term run-time automated code evolution is useful in two main situations: a need to optimize a given network service at run time, that cannot be satisfied by just optimizing service parameters; in response to steady changes in the environment or internal errors that require modifications within already deployed code.

At the same time, autonomic networks should be able to resist disruptions (hence change), including the actions of malicious or erroneous entities which

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-3-540-32993-0_29

I. Stavrakakis and M. Smirnov (Eds.): WAC 2005, LNCS 3854, pp. 13–28, 2006.

try to disturb the network's functional blocks in any possible way. Ideally, these blocks would react by detecting and defeating such attacks, and would then recover and heal themselves to continue providing the required services. In case of failures, alternative service blocks would replace the non-functioning ones in a reactive and non-supervised way.

With these problems in mind (simultaneous pressure to evolve and the requirement to resist changes) we describe in this paper our framework for protocol evolution based on genetic programming. We concentrate on two research directions: the first one is to automatically select combinations of protocol modules adapted to given network conditions; the second is the automatic synthesis of new protocols optimized for a specific context. The contribution of this paper is to show the feasibility of automatic network software selection based on service agnostic target functions. This result is based on the introduction of competition at the level of functional blocks and the use of genetic algorithms to steer the selection process. We report our experimental results using simple case studies, still in a simulated, off-line environment, but with considerations and parameters intended to progressively detach the framework from the off-line simulation out into the real world. We show the feasibility of code trimming, context aware selection of protocol variants and their re-adaption to changing environments using the proposed genetic programming framework.

This paper is structured as follows: Section 2 summarizes the state of the art in program and protocol evolution techniques. Section 3 states our position and describes our framework for protocol evolution. Section 4 reports the experimental results obtained so far. Section 5 concludes the paper with our outlook for this new area.

2 State of the Art and Related Work

Automatic programming or *program synthesis* refers to any method for automated generation of a computer program that is able to solve a given problem expressed in a high-level form. Examples include variations of meta-programming, deductive program synthesis [1], and evolutionary methods such as genetic programming.

Genetic Programming (GP) [2] is a machine learning method to evolve computer programs automatically from random initial code, using genetic operations such as crossover and mutation, and evolution by natural selection ("survival of the fittest") to select the solutions that best satisfy specified criteria. GP is typically employed when the solution to a problem is not known or very difficult to program by hand.

Although GP has been mostly applied to off-line solution of problems, it has also been used to evolve new programs at run-time, in domains such as evolvable hardware [3] and robotics [4, 5]. However, to the best of our knowledge, on-line evolution of networking protocol code has not been tried yet.

In [6] genetic algorithms are applied in a decentralized way to evolve agents that provide network services. Although their work is still implemented via

simulations, their design aims at on-line evolution. Their results show that evolution can improve agent performance. However, in their scheme, the code itself does not change. They focus on the evolution of parameters that trigger certain predefined behaviors.

Protocol synthesis [7] aims to generate a valid protocol specification that satisfies a supplied service specification. A survey of synthesis methods is provided in [7]. The methods must guarantee the safety and liveness properties of the synthesized protocols, meaning that these must be guaranteed free from syntactic, logical and semantic design errors. Since these methods must guarantee error-free code, they are still not feasible for on-line evolution.

Examples of machine learning methods applied to protocol synthesis include [8, 9, 10, 11, 12]. In [8, 9] an iterative deepening search approach is used to find protocol specifications that satisfy a given set of security properties.

In [10] genetic search is used to synthesize protocol implementations from scratch. The synthesized protocols are expressed as communicating finite state machines. This research is extended in [11] and shows that relatively complex protocols can be synthesized in this way, and in certain cases these protocols can even outperform a reference protocol designed and validated by human beings. However in most cases the fitness of synthesized protocols is significantly lower than the reference protocol.

In [12] an evolutionary method to synthesize communication protocols is proposed. Similar to [10, 11], it also synthesizes finite state machines. Moreover it includes a method to derive a set of input/output training sequences that assures semantic correctness of the generated protocol. They show that optimum protocols can be generated for the simple case of a connection establishment task.

In most of the existing work, protocol synthesis is regarded as a protocol engineering method to be applied at the design phase. In contrast, we are investigating protocol synthesis as a tool for automated protocol evolution, to be incorporated as part of the tasks that an autonomic network must handle during run-time, on a routine basis.

3 Evolving Communication Protocols

The main premise underlying our work is that software in an autonomic network must be *self-modifying*. If the software was not self-modifying, it would mean that humans had to cater for the software's adaption every time that a case is encountered which was not anticipated at design time. Our aim is to find a framework where software self-modification is carried out in a goal oriented and non-disruptive way. Hence, we seek a mechanism which is agnostic to which function it adapts as long as the mechanism is capable of steering the whole network into optimal configurations.

We envisage different levels at which self-modification of software takes place and different time scales at which such modifications can happen. In order to cope with the constraints of a realistic run-time environment, we aim first at

optimizing existing working protocol code, as opposed to full protocol synthesis from scratch. A first step, aimed at a shorter time scale, is the configuration of function blocks, where the challenge consists in selecting the right combinations from ready-made modules. Today, this is mostly controlled by standardization process and interoperability tests. Although several systems able to dynamically reconfigure software have been proposed, for instance [13, 14], most of these systems still rely on humans to program exactly what kind of reconfiguration should be performed under which circumstances. Considerable effort has also been spent on configurable protocol stacks [15, 16] but here again the reconfigurations were not fully autonomic.

In the future we imagine that a network "settles" by itself on different protocol sets without having humans to intervene. For example, depending on the available hardware, different "stack profiles" could be selected for sensors, PCs or core routers. This selection process is also applicable at finer time scales where for example an ad hoc network can switch among different routing algorithms, depending on the current topology. Another example would be the downloading of networking code, as exemplified by instantiating TCP flavors inside a TCP connection [17], where end nodes have to settle on the optimal combination of options.

At a longer time scale, these self-modification scenarios could in principle be extended down to the level of single instructions where the autonomic network would have the power to create new implementation variants, instead of just manipulating coarse grained functional blocks. At first, these new variants would emerge out of existing implementations. Eventually, full protocol synthesis from scratch, at the level of single instructions, could become possible, leading to fully autonomic networks.

3.1 Resilience and Competition

For such an autonomic selection process to work we need a modus operandi that permits adaption (medium time scale) as well as evolution (long term). Adaption relates to the configuration of existing functionality while evolution refers to the modification of old and generation of new functions. We believe that two attributes of such a system are key for its viability: resilience and competition.

The network must start with *inherent resilience*, otherwise there is a risk that (malicious or erroneous) function blocks can be inserted that disrupt the network's operation. In other words: adaption and evolution have to be activities that are running in parallel with the network and which, in the worst case, may temporarily disturb the network but cannot inhibit its operation.

The second attribute is *competition*: the autonomic network operates in a constant optimization mode where it picks those function blocks and code variants which are best suited.

Both attributes are currently implemented by having humans performing the adaption and evolution, and by writing and selecting those software bundles which provide the best value. Often, this human activity is not solely based on detailed analysis but also includes a simple trial–and–error strategy. Our goal is to rely on the later selection process only and to provide an environment where

new functionality or function profiles can be evaluated and selected without disrupting the network.

3.2 Software Hardening and Genetic Programming

We have started to explore the feasibility of self-modifying communication software by demonstrating protocol resilience, where protocol implementations can survive the removal of an arbitrary code line [18]. In the current paper we explore genetic programming as a tool for modifying, recombining and erasing protocol modules. Other machine learning methods or heuristics could also be envisaged, for example, as has been demonstrated for the synthesis of security protocols [8, 9]. However, plain genetic programming lends itself for our project because it is agnostic to the functions adapted, and naturally extends to the finer grained code evolution that enables long-term synthesis and evolution.

Another choice we have made relates to the execution environment for the protocol software, which should be amenable to genetic programming. Sequential code, for example, is less suitable than a "chemical soup of rules" execution model [19, 20] because the executability of a linear code sequence depends on almost each of its instructions. For our experiments we are currently using our "Fraglets" chemical model [21], which also permits to express code mobility e.g., for evolving code deployment logic. Section 3.5 gives a quick overview of the Fraglet model and describes its useful properties which make it our model of choice for protocol synthesis and evolution.

3.3 A Framework for Automated Code Steering

Ideally, a software environment for an autonomic network should feature continuous adaption and evolution: Alternative code variants should co-exist in parallel with the currently best selection of protocol implementations. In terms of code steering, there would be a mechanism in place for on-line evaluation and selection of the alternatives. This on-line evolution has to be a continuously ongoing process that is decentralized and asynchronous, working on each node and at many levels inside the graph of functional modules.

Figure 1 shows a conceptual model of how resilience and competition work together to enable the automatic evolution of protocol implementations and configurations. Applications (or any client protocol) delegate service provisioning to a resilient protocol implementation, and from time to time or in parallel give a chance to test candidates. Based on their performance, new service implementation variants can increase their chance to be selected a next time. Service variations do include different ways of combining sub-services. Because the evaluation and selection mechanism takes into account the overall performance of a service implementation, it will give preference to the service with the most optimal internal composition and configuration of sub-services.

Our current implementation of the model of Fig. 1 is still limited to off-line evolution, i.e. to the case of synchronous evaluation and selection, so there are

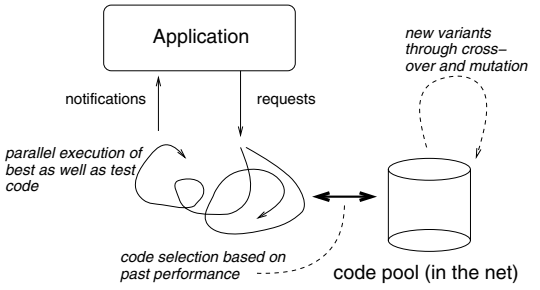

Fig. 1. Conceptual framework for automatic protocol evolution

no concurrent services yet. However we plan to progressively detach it from the off-line sphere in favor of the long-term goal of on-line evolution.

3.4 Genetic Programming Set-Up for Protocol Evolution

We apply Genetic Programming to evolve communication protocols or protocol structures, which are regarded as individuals in a GP population. A major difference between our system and classical genetic programming is that our GP run starts with a population of working or partially working solutions, which may or may not be adapted to the task in question. Another difference is that our GP run is a continuous optimization process: the system must continuously adapt and readapt. This is in contrast with classical off-line GP where the system runs until a termination condition is satisfied; it then outputs the solution and stops.

The genotype is the metaphor for the protocol implementation code, and is manipulated from one generation to the next through well-known genetic operators such as crossover, mutation and cloning. The crossover operator in our set-up is a simplified implementation of the genetic concept of homologous recombination. Homologous recombination states that the exchange of genetic material can only occur between functionally compatible DNA segments, and is only triggered when the two DNA strands are completely aligned. This form of recombination preserves gene functionality, promotes genetic stability, and increases the probability of producing viable offspring. We implement this concept by dividing the protocol genotype into modules that make up the "genes" of the individual, and by allowing crossover to occur only at gene (module) boundaries and between functionally equivalent modules.

Homologous recombination is a step towards program transformations that formally maintain program properties. If the system starts with a population of programs that contain only functionally correct modules, then homologous recombination among these programs can only produce new program variations that implement similar functionality in different ways (some might be better adapted to given situations than others), but which are still functionally correct.

The fitness measure is the performance of the protocol as perceived by the applications. They reward correct behavior and punish incorrect one when detected. For instance, the score of an individual is incremented when it performs

the correct operation (e.g. successfully delivering a packet), and it is decremented when an error is detected (e.g. an acknowledgment is issued for a data item that has never been actually received). Resource consumption, in terms of memory occupied by the genotype, is proportionally penalized. Fitness evaluation also helps keeping the system controllable, as humans can steer it through applications able to translate user input into fitness functions.

We now describe the GP algorithm. For each generation, a tournament selection is held, as follows:

1. Insert each individual of the population into its execution context (i.e. connect it to its application and network environment), and run each of them for the same fixed amount of time or execution cycles.
2. Extract the fitness scores for each individual in the population.
3. Select the n_b best fit individuals and add them to the population of the new generation.
4. From the set of n_c fittest individuals, with $n_c > n_b$, select $n_p \leq n_c/2$ pairs of individuals at random.
5. Perform crossover for each pair, producing $2 \cdot n_p$ new output code streams, which are then added to the pool of new generation individuals.
6. If mutation is enabled, select a small number n_m of individuals at random within the set of n_c fittest, and perform a mutation on each of them. Add the resulting individuals to the population of the next generation.

Traditional genetic programming models perform an off-line genetic search in which production of offspring is synchronous and fitness evaluation is centralized. Our current experiments are still limited to an off-line set-up, since we first need to demonstrate the basic viability of an automatic selection process.

3.5 Fraglets

The Fraglet paradigm [21] has been proposed as part of our search for feasible ways to achieve automated synthesis of protocol implementations. It is an instance of Gamma systems [19, 20], a chemical model where "molecules" interact with each other or undergo some internal transformation. A fraglet is a string of symbols $[\, s_1 : s_2 : \ldots : s_n \,]$ representing data and/or protocol logic. It is a fragment of a distributed computation, that may be carried in packets or stored inside a network node. 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. The fraglet instruction set contains two types of actions: transformation of a single fraglet, and "chemical reaction" between two fraglets. The instruction set is described in [21, 18], along with examples of processing and protocol functions. Table 1 summarizes the reaction and transformation rules used in the examples of Section 4.

The fraglets model has many relevant properties that must be highlighted in connection with automated protocol synthesis and evolution. First of all, any string of symbols is a valid fraglet, therefore fraglets can be split at arbitrary

Table 1. Fraglet reaction and transformation rules

Reaction	*Input*	*Output*	*Semantics*
`match`	$[\ match : s : tail_1\ ]$, $[\ s : tail_2\ ]$	$[\ tail_1 : tail_2\ ]$	concatenates two fraglets with matching tags
`matchp`	$[\ matchp : s : tail_1\ ]$, $[\ s : tail_2\ ]$	$[\ tail_1 : tail_2\ ]$ $[\ matchp : s : tail_1\ ]$	persistent match (preserves `matchp` rule)
Transf.			
`dup`	$[\ dup : t : u : tail\ ]$	$[\ t : u : u : tail\ ]$	duplicates a symbol
`exch`	$[\ exch : t : u : v : tail\ ]$	$[\ t : v : u : tail\ ]$	swaps two symbols
`split`	$[\ split : t : \ldots : * : tail\ ]$	$[\ t : \ldots\ ]$, $[\ tail\ ]$	breaks fraglet at $*$ position
`send`	${}_A[\ send : B : tail\ ]$	${}_B[\ tail\ ]$ (unreliably)	sends fraglet from A to B
`wait`	$[\ wait : tail\ ]$	$[\ tail\ ]$ (after interval)	waits a predefined interval
`nul`	$[\ nul : tail\ ]$	$[\ ]$	fraglet is removed

places and merged with other fraglets to produce different code. A second property is the ability to express code and data in a uniform way. Code is manipulated just like any other form of data, and it is easy to express rules that generate and delete code from the running pool. A third aspect is the ability to express code mobility in a natural way: any fraglet can be regarded as either a set of packet header tags that can be processed by a header processing engine, or as a program fragment that is executed at a given node. This facilitates the dynamic deployment of new code logic.

A fourth property of the fraglet environment stems from its roots in Gamma systems: it enables programs to be expressed in a highly parallel way that is very close to their specification, without artificial sequentiality constraints. This is relevant for automated program synthesis and evolution, in two ways: first, this parallelism can be used to produce resilient programs as shown in [18], which tolerate the loss of parts of their code stream, due to fallback alternatives running in parallel. This can be used to diminish the impact of malfunctioning code. Secondly, the fact that programs are relatively compact and close to their specification could open up potential avenues for deterministic synthesis techniques based on specification.

4 Experiments

We have performed a few experiments using the fraglet environment to verify whether software configurations can adapt to their environment, by the mere application of generic and service agnostic GP methods. We start with a description of the protocols involved in the experiment (Section 4.1), and then describe the results for three experiments: testing the capacity to eliminate superfluous code (Section 4.2), adaptation to the environment (Section 4.3), and re-adaptation (Section 4.4).

4.1 Protocol Implementations

A simple case is considered where a reliable delivery service must be provided over different channel characteristics. The task is to transmit all packets from

the client application, with acknowledgment of correct delivery. Two types of underlying transmission channels are considered:

- *Perfectly reliable channel*: In this case, the protocol does not need to retransmit packets. A simple implementation of this in fraglets is the confirmed delivery protocol (CDP) presented in [21]. It simply transmits a given payload from node A to node B and returns an acknowledgment from B to A.
- *Unreliable channel*: In this case, the protocol must retransmit lost packets. A reliable delivery protocol (RDP) has been implemented for this purpose. It takes an input payload from the application, sends it to the destination, stores a copy locally, and sets a waiting timer. When the timer expires, and the corresponding local copy of the information is still stored, the packet is retransmitted. When an acknowledgment is received, the local copy is destroyed; this cancels any pending retransmissions scheduled for the item. For simplification, no losses from sink to source are modeled.

Each protocol is encoded as a fraglet genotype made up of constituent modules or genes. The genotype is the concatenation of all the modules (and their constituent fraglets) that implement the protocol. Each module starts with an *"m"* marker followed by the module name.

Fig. 2 shows the fraglet code for CDP, both sender and receiver sides. When presented with an application payload of the form $_A[data : payload]$, the first *matchp* rule in the *send* module will be activated, and the resulting reaction will produce a rule $_A[send : B : deliver : payload]$, which will send the fraglet $[deliver : payload]$ to B, where the *deliver* tag will cause *payload* to be delivered to the application. The application will respond by injecting a $_B[ack]$ fraglet, which will react with the *matchp* rule of the *receive* module, causing the *ack* to be delivered to the source application on node A. Note that the *deliver* tag can be implemented as a predefined rule that takes the tail symbol string out of the fraglet environment (towards an external application), or can be caught by a $[matchp : deliver : ...]$ rule as part of a fraglet application.

m send	**m receive**
$_A[matchp : data : send : B : deliver]$	$_B[matchp : ack : send : A : deliver : ack]$

Fig. 2. CDP implementation in fraglets

The RDP implementation is shown in Fig. 3. It has exactly the same interface with the application as CDP, so that both protocols can be interchanged in a transparent way. A $[data : payload]$ fraglet injected by the application activates the *send* module, producing two fraglets: $[retransmit : payload]$ and $[mack : payload]$. The first one triggers a retransmission loop (*retransmit* module). The second one triggers a series of reactions which produce a new rule able to treat an incoming *ack* and cancel any corresponding retransmission.

Several variants of CDP and RDP have been implemented to make up a reasonably sized initial population for the GP run. Figures 2 and 3 show examples

m send
[*matchp* : *data* : *dup* : *data*3]
[*matchp* : *data*3 : *exch* : *data*2 : *mack*]
[*matchp* : *data*2 : *exch* : *data*1 : ∗]
[*matchp* : *data*1 : *split* : *retransmit*]
[*matchp* : *mack* : *exch* : *mack*5 : *nul*]
[*matchp* : *mack*5 : *exch* : *mack*4 : ∗]
[*matchp* : *mack*4 : *dup* : *mack*3]
[*matchp* : *mack*3 : *exch* : *mack*2 : *wait*]
[*matchp* : *mack*2 : *exch* : *mack*1 : *split*]
[*matchp* : *mack*1 : *match* : *ack* : *split* : *deliver* : *ack* : ∗ : *match*]

m retransmit
[*matchp* : *retransmit* : *dup* : *t*91]
[*matchp* : *t*91 : *exch* : *t*92 : *t*94]
[*matchp* : *t*92 : *exch* : *t*93 : ∗]
[*matchp* : *t*93 : *split* : *transmit*]
[*matchp* : *t*94 : *dup* : *t*95]
[*matchp* : *t*95 : *exch* : *t*96 : *retransmit*]
[*matchp* : *t*96 : *dup* : *t*97]
[*matchp* : *t*97 : *exch* : *t*98 : ∗]
[*matchp* : *t*98 : *split* : *wait* : *match*]
[*matchp* : *transmit* : *send* : *B*]

Fig. 3. RDP implementation in fraglets (sender side)

of correct implementations. Other correct variants are also present in the experiments, as well as variants that introduce arbitrary delays, consume more memory, contain useless code segments, pollute the code pool with byproduct debris of reactions, and so on.

Crossover by homologous recombination is implemented by swapping modules of the same name in different protocol implementations. Since the interface of each module is the same regardless of its internal implementation, modules are compatible and crossover produces viable individuals. Mutation is applied with a low probability, changing a symbol at random in the fraglet pool.

4.2 Stripping Protocol Implementations

In this first baseline experiment we test whether the system is able to strip exceeding code, by eliminating garbage that is arbitrarily added to the programs. We take the CDP implementation and add several modules, some of which are empty, and some which perform random but non-disruptive actions consuming CPU cycles.

We generate 10 such "polluted" individuals, and perform repeated GP runs of 50 generations each, and $n_b = 4$, $n_c = 8$, $n_p = 3$, $n_m = 0$. A typical result from these runs is that roughly 75% of the garbage modules are eliminated. In a sample run, a relatively clean individual (with a single garbage module remaining) emerges around the second generation, and progressively propagates to the rest of the population. By the 7th generation, all individuals have a single garbage module. In this example the system does not improve beyond that, because all the individuals have the same garbage module, therefore homologous crossover is not able to eliminate it.

4.3 Adaptation

The goal of this experiment is to verify whether a mixed population of protocols is able to adapt to a given environment. Our mixed population is composed of eight CDP and eight RDP variants. These are alternative implementations of the same functionality. Some of them are perfect with no known bugs, others

are deliberately made inefficient to different degrees, for instance, by not retransmitting packets correctly, or retransmitting too much, or spending a lot of time on bogus tasks.

We insert this population into two GP runs. In the first run, the population faces a reliable channel with no packet loss. In the second run a rather lossy channel (25% packet loss) is introduced. For each run we choose $n_b = 6$, $n_c = 14$, $n_p = 4$, $n_m = 2$. This results in a population size of $N = n_b + 2 \cdot n_p + n_m = 16$ individuals per generation, which is the same size as the original (hand-made) population.

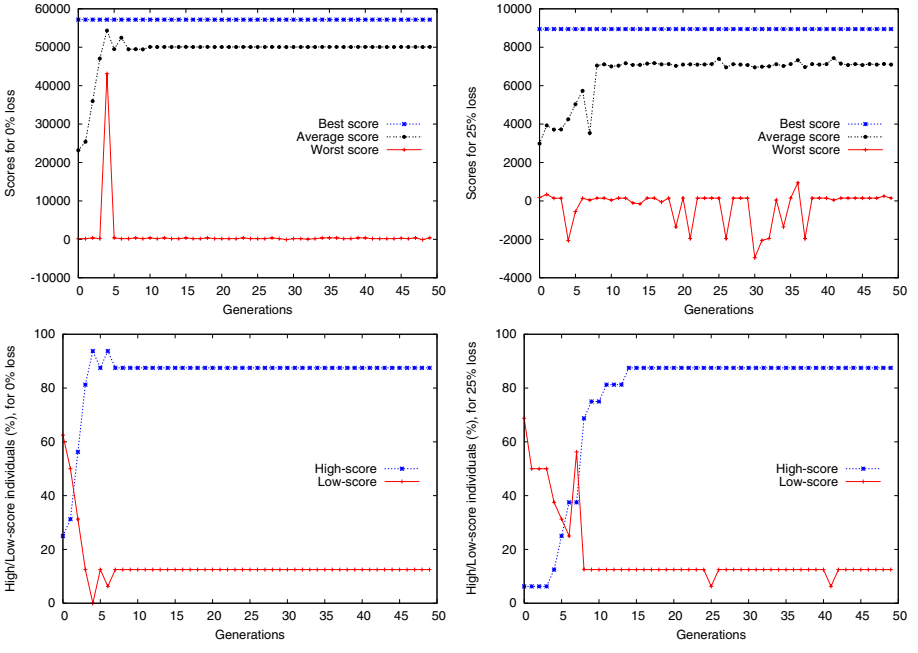

Fig. 4. Absolute scores and percentage of high/low scores for different packet loss rates

Figure 4 shows the adaptation of the initially mixed population to these two loss environments. The upper part shows the fitness scores for the different link loss rates, and the lower part shows the percentage of high and low-score individuals. A high-score individual is an individual that has achieved a score equivalent to at least 80% of the best score from its generation. A low-score one scores less than 40% of the best of its generation.

For the non-lossy channel (Fig. 4 left), the population starts with a low average score, but after a few generations most of the individuals have a score close to the best, and the percentage of individuals with very low score is small. In this case, the best individual is also the optimum (hand-designed), and the GP selection process succeeds to keep it in the population through the successive generations. After four or five generations the retransmission code is eliminated, and the surviving individuals are all instances of CDP.

In the lossy channel the retransmission code spreads very quickly through the entire population: all the individuals contain it after the first couple of generations. In Fig. 4 (top right) we can notice that the best score achieved by RDP is much lower than its equivalent in CDP. This is because the retransmit logic and associated timers consume execution cycles. Since all the individuals are allowed to consume the same amount of cycles, the simple code achieves much higher score. The adaptation to the environment can be observed in Fig. 4 (bottom right): after roughly 15 generations, more than 80% of the population is made up of high-score individuals. At the same time, the number of low-score individuals is reduced to a minimum.

In both lossy and non-lossy cases, mutations are mostly responsible for these low-performance individuals. The purpose of mutations is to introduce genetic variability. However, it is well known that most mutations are harmful. In our case, mutations are kept in the system in order to test its capacity to produce new code, and its resilience to potentially disrupting code. The production of new useful code has not been verified in such short runs though. On the other hand, the fact that the system can still adapt in spite of harmful mutations is an indication that resilience at the population level is possible even with the high rate of mutation chosen ($n_m/N = 12.5\%$). However this system is obviously not perfect. There are still clients affected by low-performance individuals: resilience is not achieved at the individual level. Furthermore, as it adapts, the population also loses genetic variability (this will be discussed in the next section). We believe this sort of drawback can be diminished if resilient individuals incorporating redundancy are used in place of the current non-resilient ones.

4.4 Re-adaptation

In this experiment we investigate the capacity of a population to readapt to an environment different from the one where it has originally evolved. We inject a population evolved in a 25% loss environment into a no-loss and vice-versa, and repeat the GP run with the same parameters as described in Section 4.3.

Figure 5 shows the obtained scores. These results clearly show that the population is not able to readapt. The lost retransmission modules cannot be recreated in such a short time by genetic operators only. The homologous crossover used only recombines existing modules, and mutations of individual symbols is simply a too slow and randomized process. The search space for the solution is far too vast, even though GP has shown to remarkably focus the search when compared to pure random search. For example, in the RDP example of Fig. 3, there are about 20 different symbols that may be placed at about 100 positions, leading to a search space of size 20^{100}. This is still too vast for short-term on-line GP. A similar problem may also occur in nature, when genetic variability is lost in small populations adapted to a fairly stable environment.

Nevertheless, if we inject a single optimally adapted individual in the population, it instantly redeploys and the entire population readapts. This can be observed in Fig. 6. After about 15 generations, more than 80% of its individuals achieve scores comparable to those of the best individuals of Section 4.3.

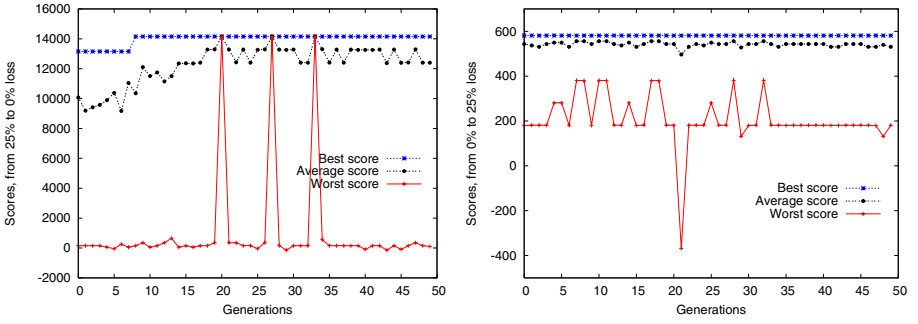

Fig. 5. Scores for two different re-adaptation situations: Left: from 25% loss to 0% loss. Right: from 0% loss to 25% loss.

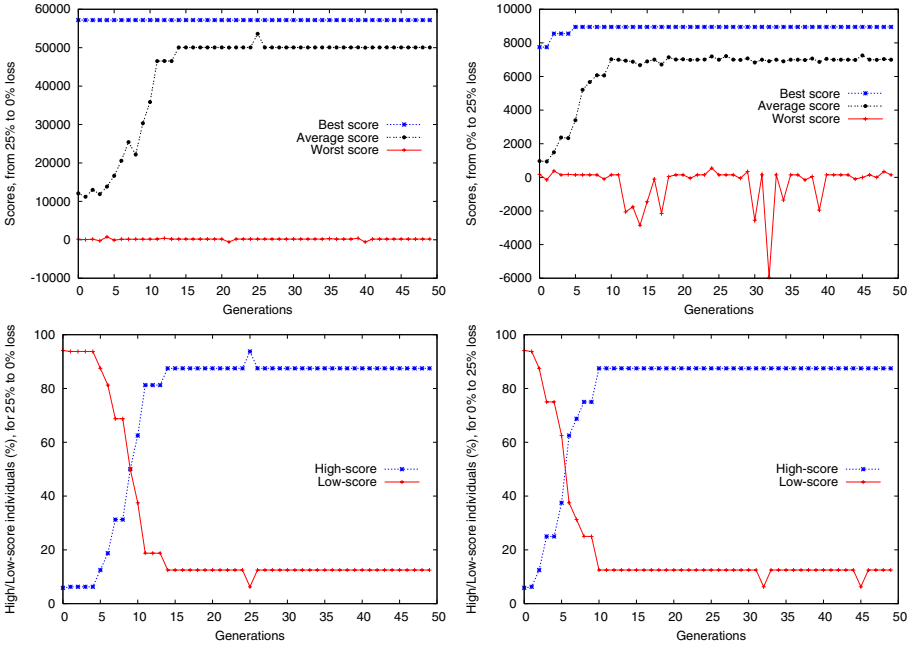

Fig. 6. Inserting a single adapted individual. Scores (top) and percentage of high/low scores (bottom) for different adaptation situations: Left: from 25% loss to 0% loss. Right: from 0% loss to 25% loss.

4.5 Discussion

We can extract several lessons from these early experiments. We first discuss the aspects related to genetic operators and other GP parameters. We then discuss future issues of resilience and on-line evolution.

We have modeled homologous recombination which is generally overlooked in GP. By restricting crossover to functionally compatible genes only, we have a high probability of producing viable individuals. In a few earlier experiments we had tried crossover at arbitrary points, and the result was poor score evolution combined with the well-known code bloat phenomenon in GP [2, 22], in which code tends to grow across generations, leading to large, inefficient programs in the long run. A widespread theory to explain the phenomenon says that GP code accumulates *introns* [2], i.e. portions of code that serve no functional purpose. These introns would then act as a protection against destructive crossover, as the probability of crossover points falling inside an intron (and therefore not breaking existing useful functionality) increases with the percentage of introns in the individual. Experimental results [22] show that code growth occurs even when crossover within introns is not allowed. However these results are valid only for tree-based GP, which is not our case. Anyway, independently of the actual causes of code bloat in a general sense, in our experiments the phenomenon disappeared as soon as we introduced homologous recombination.

However, homologous recombination in a limited population of simple individuals with few genes, as shown in the experiments, leads to low genetic variability, and after a few generations most of the variability is lost.

Mutation is usually regarded as the main source of genetic variability in GP populations [2]. However, the benefits of mutation can only be observed at the very long run, since most mutations are lethal. In our short-run experiments, we have not been able to observe really productive mutations. We have to interpret these very preliminary results with caution; nevertheless, they seem to indicate that new, more intelligent techniques for evolving populations of genetic protocols need to be devised to make on-line evolution a reality.

The parameters of a GP run clearly have an impact on the evolutionary process. Adjusting these parameters is a well-known difficult problem in GP. Some researchers have inserted GP parameters into the genotypes evolved such that the best combination of parameters can also emerge from the evolutionary algorithm itself. This is a path we intend to explore in our future work.

In our current experimental set-up, fitness evaluation still has a centralized component. This prevents the emergence of cheat programs, e.g. programs that lie about transmitted or acknowledged packets. Fitness evaluation is a non-trivial issue in a real distributed on-line environment. Perhaps redundancy and reputation mechanisms could be combined to to provide a safe and reliable way to evaluate the behavior of protocols at run-time.

The next immediate step towards on-line evolution that we are starting to investigate is how to combine our previous resilience work [18] with genetic programming in order to add resilience at the level of individuals, as opposed to the level of entire populations as described in the experiments above. Each protocol is modeled as tuples of redundant genetic code. This should in principle improve resilience, and help preserving genetic variability in small populations.

5 Conclusions and Outlook

In this paper we propose an intrinsic approach to the automated evolution of network software. The goal is to enable automatic code deployment, self-configuration of functional modules and even automatic synthesis of protocol implementations in an autonomic network. We argue that the automated selection of protocols becomes feasible if the networking code is *resilient* such that we can have *competing* protocol variants running in parallel.

Using a concept known as homologous recombination, we have carried out exploratory adaption experiments using genetic programming. They show that a networking system can automatically and gradually evolve depending on the environment it is confronted with, provided that a minimum variability of code instances is kept. This observation relates both to identifying an optimal protocol implementation for a given context, as well as to finding the most efficient combination of several software modules.

A more complex task, that has yet to be demonstrated, is an on-line version where software evolution is a continuous activity. Our experiments have provided first insights on the obstacles that have to be overcome: For instance, fitness evaluation in a decentralized and competitive environment is a non-trivial issue. Another fundamental issue is to devise new and potentially correctness preserving genetic operators beyond homologous crossover which are able to evolve genuine new code for unforeseen situations.

References

1. Manna, Z., Waldinger, R.: Fundamentals of Deductive Program Synthesis. IEEE Transactions on Software Engineering **18** (1992) 674 – 704
2. Banzhaf, W., Nordin, P., Keller, R.E., Francone, F.D.: Genetic Programming, An Introduction. ISBN 155860510X. Morgan Kaufmann Publishers, Inc. (1998)
3. Sipper, M., Sanchez, E., Mange, D., Tomassini, M., Perez-Uribe, A., Stauffer, A.: A Phylogenetic, Ontogenetic, and Epigenetic View of Bio-Inspired Hardware Systems. IEEE Transactions on Evolutionary Computation **1** (1997)
4. Andersson, B., Svensson, P., Nordin, P., Nordahl, M.: On-line Evolution of Control for a Four-Legged Robot Using Genetic Programming. In: Real-World Applications of Evolutionary Computing – EvoWorkshops 2000. Springer-Verlag LNCS 1803, Edinburgh, Scotland (2000) 319–326
5. Steels, L.: Emergent functionality in robotic agents through on-line evolution. In: Proceedings of AlifeIV, Cambridge, MIT Press. (1994)
6. Nakano, T., Suda, T.: Adaptive and Evolvable Network Services. In: Proc. Genetic and Evolutionary Computation Conference (GECCO-2004). Springer-Verlag LNCS 3102 (2004) 151–162
7. Probert, R.L., Saleh, K.: Synthesis of Communication Protocols: Survey and Assessment. IEEE Transactions on Computers **40** (1991) 468 – 476
8. Perrig, A., Song, D.: A First Step towards the Automatic Generation of Security Protocols. In: Proc. Network and Distributed System Security (NDSS 2000). (2000)
9. D. Song, A.P., Phan, D.: AGVI – Automatic Generation, Verification, and Implementation of Security Protocols. In: Proc. 13th Conference on Computer Aided Verification (CAV 2001). (2001)

10. Sharples, N., Wakeman, I.: Protocol construction using genetic search techniques. In: Real-World Applications of Evolutionary Computing – EvoWorkshops 2000. Springer-Verlag LNCS 1803, Edinburgh, Scotland (2000)
11. Sharples, N.: Evolutionary Approaches to Adaptive Protocol Design. PhD dissertation, University of Sussex, UK (2001)
12. Araújo, S.G., Pedroza, A.C.P., Mesquita, A.C.: Evolutionary Synthesis of Communication Protocols. 10th International Conference on Telecommunications (ICT 2003) **2** (2003) 986–993
13. Whisnant, K., Kalbarczyk, Z.T., Iyer, R.K.: A system model for dynamically reconfigurable software. IBM Systems Journal **42** (2003) 45–59
14. Ruf, L., Keller, R., Plattner, B.: A Scalable High-performance Router Platform Supporting Dynamic Service Extensibility On Network And Host Processors. In: Proceedings of the 2004 ACS/IEEE International Conference on Pervasive Services (ICPS'2004), Beirut, Lebanon (2004) 19–23
15. Hutchinson, N.C., Peterson, L.L.: The x-Kernel: An architecture for implementing network protocols. IEEE Transactions on Software Engineering **17** (1991) 64–76
16. Plagemann, T.: A Framework for Dynamic Protocol Configuration. PhD dissertation, Swiss Federal Institute of Technology Zurich, Zurich, Switzerland (1994)
17. Patel, P., Wetherall, D., Lepreau, J., Whitaker, A.: TCP Meets Mobile Code. In: Proc. of the Ninth Workshop on Hot Topics in Operating Systems. IEEE Computer Society. (2003)
18. Tschudin, C., Yamamoto, L.: A Metabolic Approach to Protocol Resilience. In: Proc. 1st Workshop on Autonomic Communication (WAC 2004). Springer-Verlag LNCS 3457, Berlin, Germany (2004) 190 – 205
19. Banâtre, J.P., Métayer, D.L.: Gamma and the Chemical Reaction Model. Internal Publication PI-984, INRIA, France (1996)
20. 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
21. Tschudin, C.: Fraglets - a Metabolistic Execution Model for Communication Protocols. In: Proc. 2nd Annual Symposium on Autonomous Intelligent Networks and Systems (AINS), Menlo Park, USA (2003)
22. Luke, S.: Issues in Scaling Genetic Programming: Breeding Strategies, Tree Generation, and Code Bloat. PhD dissertation, University of Maryland, USA (2000)