

# Evolving Proactive Aggregation Protocols

Thomas Weise, Michael Zapf, and Kurt Geihs

University of Kassel, Wilhelmshöher Allee 73, D-34121 Kassel, Germany  
{weise, zapf, geihs}@vs.uni-kassel.de  
<http://www.vs.uni-kassel.de>

**Abstract.** We present an approach for the automated synthesis of proactive aggregation protocols using Genetic Programming and discuss major decisions in modeling and simulating distributed aggregation protocols. We develop a genotype, which is an abstract specification form for aggregation protocols. Finally we show the evolution of a distributed average protocol under various conditions to demonstrate the utility of our approach.

## Preview

This document is a preview version  
and not necessarily identical with  
the original.

<http://www.it-weise.de/>

## 1 Introduction

Genetic Programming has some popular application areas like the synthesis of analog electrical circuits, cellular automata, and data mining. In larger networks, especially sensor networks or MANETs, the design of protocols for distributed processing may become a challenge. We believe that employing Genetic Programming for the automatic synthesis of protocols reveals a lot of future potential for designing distributed systems [1–3].

In order to unleash this potential, a clear and structured approach is needed. In this paper, we stepwise exercise such an approach on the example of automated distributed aggregation protocol synthesis.

Aggregation functions with their ability to summarize information in a certain, user-specified way are a very important building block for distributed applications [4]. We illustrate its utility in sensor networks [5] in an initial example scenario in Section 2. This example helps to clarify the problem domain and the required features of possible solutions. We then select a suitable Genetic Programming technique that can be extended to suffice these needs. The next step (taken in Section 3) is to derive suitable models for the entities in the problem

domain. This means modeling the *relevant* properties of both, the nodes that will run the protocols and the network itself. Now the structure of the solution candidates and how they have to be simulated in order to determine their fitness can be defined, as done in Section 4. Section 5 shows results from experiments which demonstrate the applicability of the approach before we conclude this article in Section 6.

## 2 The Scenario

### 2.1 Gossip-Based Aggregation

Jelasy, Montresor and Babaoglu [6] propose a simple yet efficient type of proactive aggregation protocols [7]. Its basic assumption is that each node in a network holds one numerical value  $x$ . This value represents the information about the node or its environment that should be aggregated, for example the current work load. The task of an aggregation protocol is to provide all nodes in the network with an up-to-date estimate of the aggregate function  $\alpha(\mathbf{x})$  of the vector of all values  $\mathbf{x} = (x_p, x_q, \dots)$ . Of course, we cannot compute  $\alpha$  directly since  $\mathbf{x}$  is distributed over the network.

The nodes hold local states  $s$  (containing  $x$ ) which they can exchange via messages. Therefore, each node regularly picks a communication partner with the function *getNeighbor()*. Once in each  $\delta > 0$  time units, at a randomly picked time, a node  $p$  selects a neighbor  $q$ . Both partners exchange their information and update their states with the *update* method:  $p$  calls *update*( $s_p, s_q$ ) and  $q$  invokes *update*( $s_q, s_p$ ). *update* is defined according to the aggregate that we want to be computed.

### 2.2 The Distributed Average

Imagine a network of distributed temperature sensors, carrying a little display visible to the public. The temperatures measured locally will fluctuate because of wind or light changes. Thus, the displays should not only show the temperature measured by the sensor node they are directly attached to, but also the average of all temperatures measured by all nodes. The network needs to execute a distributed aggregation protocol in order to estimate that average. If we choose a gossip-based average protocol, each node will hold a state variable containing its local estimation of the mean. The *update* function, receiving the local approximation and the estimate of another node, returns the mean of its inputs.

$$update_{avg}(s_p, s_q) = \frac{s_p + s_q}{2} \quad (1)$$

If two nodes  $p$  and  $q$  communicate with each other, the new value of  $s_p$  and  $s_q$  will be  $s_p(t+1) = s_q(t+1) = 0.5 * (s_p(t) + s_q(t))$ . The sum – and thus also the mean – of both states remains constant. Their variance, however becomes 0 and so the overall variance in the network gradually decreases.

### 2.3 Why synthesize aggregation protocols?

There are three use cases for an automated aggregation protocol synthesis:

- We may already know a valid aggregation protocol but want to find an equivalent protocol which has advantages like faster convergence or robustness in terms of input volatility. This case is analogous to finding arithmetic identities in symbolic regression.
- We do not know the aggregate function  $\alpha$  nor the protocol but have a set of sample data vectors  $\mathbf{x}_i$  (maybe differing in dimensionality) and corresponding aggregates  $y_i$ . Using Genetic Programming we attempt to find an aggregation protocol that fits to this sample information.
- The most probable use case is that we know how to compute the aggregate locally with a given  $\alpha$  function but want to find a distributed protocol that does the same. In order to automate this transformation, we use  $\alpha$  to create sample data sets and then apply the approach of the second use case.

### 2.4 Symbolic Regression

Our goal is to provide a framework which allows such an automated translation of a (local) aggregate function  $\alpha$  into a proactive, gossip-based aggregation protocol. For this purpose we extend Koza's technique of *symbolic regression* [8].

Regression means finding a function  $f^* : \mathbb{R}^m \mapsto \mathbb{R}$  that approximates an unknown relation  $\varphi$  of one dependent variable  $y \in \mathbb{R}$  to  $m$  independent variables  $x_1, x_2, x_3, \dots, x_m$  by analyzing a given set of sample data  $S = \{(x_{1,i}, x_{2,i}, \dots, x_{m,i}, y_i)\}$ . Traditional approaches like linear regression define a parametric curve  $f_{\mathfrak{P}}$  and then minimize the mean square error MSE of the curve to the data samples by optimizing the parameters  $\mathfrak{P}$ .

$$MSE(f) = \frac{1}{|S|} \sum_{i=1}^{|S|} [y_i - f_{\mathfrak{P}}(x_{1,i}, \dots, x_{m,i})]^2 \quad (2)$$

Symbolic regression does not limit the solution to a given form, as mentioned in numerous works [8–10]. Here, mathematical expressions are represented as tree structures. Nodes are mathematical functions and their child nodes are their parameters. Real constants and the independent variables  $x_1 \dots x_m$  act as leaves. The (functional) objective function is usually the mean square error  $\gamma_1(f) \equiv MSE(f)$  which turns symbolic regression into a maximum likelihood estimation method [11].

In the following sections we will show how this approach can be extended in order to allow the evolution of proactive aggregation protocols.

## 3 Modeling

### 3.1 Network Model

An important aspect of communication is how the nodes select their partners for the data exchange. Jelasity, Montresor, and Babaoglu have shown that different

*getNeighbor* methods influence the convergence speed of the protocols [6]. A suitable partner selection method leads to fast convergence, speeding up the simulations used for evaluating the protocols during the evolution significantly.

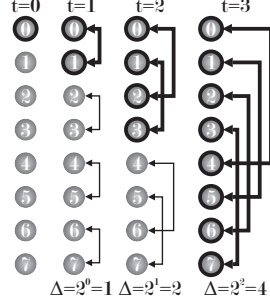


Fig. 1: pair-based dissemination

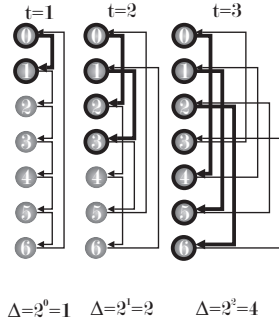


Fig. 2: generalized dissemination

---

**Algorithm 1:**  $\gamma_1(u, e, r) = evalAggProtocol(u, m, T)$

---

**Input:**  $u$ , the evolved protocol *update* function  
**Input:**  $m$ , the number of nodes in the simulation  
**Input:**  $T$ , the maximum number of simulation steps  
**Output:**  $\gamma_1(u, e, r)$ , the sum of all square errors

```

1 begin
2    $d \leftarrow \lceil \log_2 m \rceil$ 
3    $S(0) \leftarrow new\ n \times m\ Matrix$ 
4   // initialize with local values
5    $S(0)_{i,k} \leftarrow getInput(k, 0)$ 
6    $t \leftarrow 1$ 
7   while  $t \leq T$  do
8      $S(t) \leftarrow copyMatrix(S(t-1))$ 
9     // perform communication
10     $k \leftarrow 1$ 
11    while  $k \leq m$  do
12       $p \leftarrow (k + \Delta) \bmod m$ 
13       $S(t)_{r_j,p} \leftarrow S(t-1)_{e_j,k} \forall j = 1 \dots |r|$ 
14       $k \leftarrow k + 1$ 
15     $k \leftarrow 1$ 
16    while  $k \leq m$  do
17       $S(t)_{i,k} \leftarrow getInput(k, t)$ 
18       $S(t)_{*,k} \leftarrow u(S(t)_{*,k})$ 
19       $res \leftarrow res + (y(t) - S(t)_{o,k})^2$ 
20       $k \leftarrow k + 1$ 
21     $t \leftarrow t + 1$ 
22  return  $res$ 
23 end
```

---

For networks  $\mathcal{N}$  which have a number of nodes of  $m = |\mathcal{N}| = 2^d$ , we can specify an optimal selection scheme: In each protocol step  $t$  with  $t = 1, 2, \dots$ , we compute a value  $\Delta = 2^{t \bmod d}$ . We then build pairs in the form  $(i, i + \Delta)$ , where  $i$  is the ID number of the node. This setup is optimal in terms of convergence speed, as shown in Figure 1. The data from node 0 (marked with a thick border) spreads in the first step to node 1. In the second step, it reaches node 2 directly and node 3 indirectly through node 1. In the third protocol step, the remaining four nodes receive knowledge of the information from node 0. Now the cycle would start over again.

We can generalize this approach for networks sizes that are no powers of 2. Here, we set  $d = \lceil \log_2 m \rceil$  while still leaving  $\Delta = 2^{t \bmod d}$  and define that a node  $i$  sends its data to the node  $(i + \Delta) \bmod m$  for all  $i$  as illustrated in Figure 2.

### 3.2 Node Model

The model of nodes comprising a network is just as important as the network model itself. A node  $p$  executing a gossip-based aggregation protocol receives input in form of the locally known value (for example, a sensor reading) and also in form of messages containing data from other nodes in the network. The output of  $p$  is, on one hand, the local approximation of the aggregate value, and on the other hand the information sent to its partners in the network. The computation is done by a processor which updates the local state by executing the *update* function. The local state  $s_p$  of  $p$  can most generally be represented as a vector  $\mathbf{s}_p \in \mathbb{R}^n$  of dimension  $n$ , where  $n$  is the number of memory cells available on a node.

Until now, we have considered the states to be scalars. Generalizing them to vectors allows us to specify or evolve more complicated protocols. The state vector contains approximations of aggregate values at positions  $1 \leq i \leq n$ . It does not only serve as a container for the aggregate, but also as memory capable of accumulating information. It is probably unnecessary to exchange the complete state during the communication. Therefore we specify an index list  $e$  containing the indices of the elements to be sent and a list  $r$  with the indices of the elements that shall receive the values of the incoming messages. For a proper communication between the nodes, the length of  $e$  and  $r$  must be equal and each index must occur at most once in  $e$  and also at most once in  $r$ . Whenever a node  $p$  receives a message from node  $q$ , the following assignment will be done, with  $s[i]$  being the  $i^{\text{th}}$  component of the vector:

$$\mathbf{s}_p[r_j] \leftarrow \mathbf{s}_q[e_j] \quad \forall j = 1 \dots |r| \quad (3)$$

In the original form of gossip-based aggregation protocols, the state is initialized with a static input value which is stepwise refined to approximate the aggregate value [6]. In our model, this restriction is no longer required. We specify an index  $I$  pointing at the element of the state vector that will receive the input. This allows us to grow protocols for static and for volatile input data – in the latter case, the inputs are refreshed in each protocol step. A node  $p$  would then perform

$$\mathbf{s}_p(t)[I] \leftarrow \text{getInput}(p, t) \quad (4)$$

The function *getInput*( $p, t$ ) returns the input value of node  $p$  at time step  $t$ . With this definition, the state vectors  $\mathbf{s}$  become time-dependent, written as  $\mathbf{s}(t)$ . Finally, *update* is now designed as a map  $\mathbb{R}^n \mapsto \mathbb{R}^n$  to return the new state vector.

$$\mathbf{s}_p(t+1) = \text{update}(\mathbf{s}_p(t)) \quad (5)$$

In the simulation, we can put the state vectors of all nodes together to a single  $n \times m$  matrix  $S(t)$ . The column  $k$  of this matrix contains the state vector  $\mathbf{s}_k$  of the node  $k$ .

$$S(t) = (\mathbf{s}_1, \mathbf{s}_2, \dots, \mathbf{s}_m) \quad (6)$$

$$S_{j,k} = \mathbf{s}_k[j] \quad (7)$$

## 4 Breeding the protocol

### 4.1 Evaluation and Objective Values

The models described before are the basis of the evaluation of the aggregation protocols that we breed. In general, there are two functional features that we want to develop in the artificial evolution:

1. We want to grow aggregation protocols where the deviation between the local estimates and the global aggregate is as small as possible, ideally 0.
2. This deviation can surely not be 0 after the first iteration at  $t = 1$ , because the nodes do not know all data at that time. However, the way how received data is incorporated into the local state of a node influences the speed of convergence to the wanted value. We want to find protocols that converge as quickly as possible.

In all use cases discussed in Section 2.3, we already know either the correct aggregation values  $y_i$  or the local aggregate function  $\alpha : \mathbb{R}^m \mapsto \mathbb{R}$  that calculates them from data vectors of the length  $m$ . The objective is to find a distributed protocol that computes the same aggregates in a network where the data vector is distributed over  $m$  nodes. In our model, the estimates of the aggregate value can be found at the positions  $S_{O,*} \equiv \mathbf{s}_k[O] \forall k \in 1 \dots n$  in the state matrix or the state vectors respectively.

The deviation  $\varepsilon(k, t)$  of the local approximation of a node  $k$  from the correct aggregate value  $y(t)$  at a point in time  $t$  denotes its estimation error.

$$y(t) = \alpha((getInput(1,t), \dots, getInput(m,t))') \quad (8)$$

$$\varepsilon(k, t) = y(t) - S_{O,k}(t) = y(t) - \mathbf{s}_k[O] \quad (9)$$

We have already argued that the mean square error is an appropriate quality function for symbolic regression. Analogously, the mean of the squares of the errors  $\varepsilon$  over all simulated time steps and all simulated nodes is a good criterion for the utility of an aggregation protocol. It tangents both functional aspects subject to optimization: The larger it is, the greater is the deviation of the estimates from the correct value. If the convergence speed of the protocol is low, these deviations will become smaller more slowly by time. Hence, the mean square error will also be higher. For any evolved *update* function  $u$  we define<sup>1</sup>:

$$\gamma_1(u, e, r) = \frac{1}{T * m} \sum_{t=1}^T \sum_{k=1}^m \varepsilon(k, t)^2 \Big|_{u,e,r} \quad (10)$$

This rather mathematical definition is realized indirectly in Algorithm 1, which returns the value of  $\gamma_1$  for an evolved *update* method  $u$  and also applies the fast, convergence-friendly communication scheme discussed in Section 3.1.

<sup>1</sup> where  $|_{u,e,r}$  means “passing  $u, e, r$  as input to Algorithm 1”

### 4.2 Phenotypic Representation

We have to find a proper representation for gossip-based aggregation protocols. Such a protocol consists of two parts: the evolved *update* function and a specification of the properties of the state vector – the variables  $I$ ,  $O$ ,  $r$ , and  $e$ .

**Representation for the *update* function** The function *update* as defined in the context of our basic model for aggregation protocols receives the state vectors  $\mathbf{s}_k(t) \in \mathbb{R}^m$  of time step  $t$  as input. It returns the new state vectors  $\mathbf{s}_k(t+1) \in \mathbb{R}^m$  of time step  $t+1$ . This function is indeed an algorithm by itself which can be represented as a list of tuples  $l = [\dots, (u_j, v_j), \dots]$  of mathematical expressions  $u_j$  and vector element indices  $v_j$ . This list  $l$  is processed sequentially for  $j = 1, 2, \dots, |l|$ . In each step  $j$ , the result of the expression  $u_j$  is computed and assigned to the  $v_j$ th element of the old state vector  $\mathbf{s}(t-1)$ . In the simplest case,  $l$  will have the length  $|l| = 1$ . One example for this is the well-known distributed average protocol illustrated in Figure 3: In the single formula, the first element of  $\mathbf{s}_1(t)$ ,  $[1]$ , is assigned to  $0.5 * ([1] + [2])$  which is the average of its old value and the received information. Here the value of the first element is sent to the partner and the received message is stored in the second element, i.e.  $r = [2], e = [1]$ . The terminal set of the expressions now does not contain the simple variable  $x$  anymore but all elements of the state vectors. Finally, after all formulas in the list have been computed and their return values are assigned to the corresponding memory cells, the modified old state vector  $\mathbf{s}_k(t)$  becomes the new one  $\mathbf{s}_k(t+1)$ .

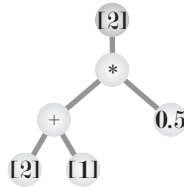


Fig. 3: A distributed average protocol.

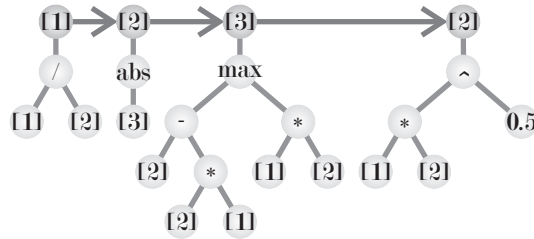


Fig. 4: The square root of the distributed average protocol.

Figure 4 shows a more complicated protocol where *update* consists of  $|l| = 4$  formulas  $[(u_1, 1), (u_2, 2), (u_3, 3), (u_4, 2)]$ . We will not elaborate deeper on these examples but just note that both are obtained with Genetic Programming. The point is that we are able to provide a form for the first part of the aggregation protocol specification that is compatible to normal symbolic regression and which hence can be evolved using standard operators.

Besides a sequence of formulas computed repetitively in a cycle, we also need an additional sequence that is executed only once, in the initialization

phase. This is needed for some other protocols than the distributed minimum, maximum, and average, which cannot assume the approximation of the estimate to be the current input value. Here, another sequence of instructions is needed which transforms the input value into an estimate which then can be exchanged with other nodes and used as basis for subsequence calculations. This additional sequence is evolved and treated exactly in the same way as the set of formulas used inside the protocol cycle.

Straightforward, we can specify a non-functional objective function  $\gamma_2$  that returns the number of expressions in both sets and hence puts pressure into the direction of small protocols with less computational costs.

**Representation for  $I$ ,  $O$ ,  $e$ , and  $r$**  Like the *update* function, the parameters of the data exchange,  $r$  and  $e$ , become subject to evolution.  $I$  and  $O$  are only single indices; we can assume them to be fixed as  $I = 1$  and  $O = 2$ . Although we could do the same with  $e$  and  $r$ , there is a very good reason to keep them variable: If  $e$  and  $r$  are built during the evolutionary process, different protocols with different message lengths ( $|e_1| \neq |e_2|$ ) can emerge. Therefore, we introduce a non-functional objective function  $\gamma_3$  minimizing the message lengths. The results of Genetic Programming will thus be optimal not only in accuracy of the aggregates but also in terms of communication costs. A good encoding scheme for  $e$  and  $r$  is a variable-length integer string (array) for each of the two. Such genomes are common and we can reuse standard operators of genetic algorithms.

### 4.3 Volatile Input Data

The specification of *getInput*( $k, t$ ) which returns the input value of node  $k$  at time  $t \in [0, T]$  allows us to evolve aggregation protocols for static as well as for volatile input. Traditional aggregation protocols are only able to deal with static inputs [6], having good convergence properties, as illustrated in Figure 5.

They would need to be restarted in a real application from time to time in order to provide up-to-date approximations of the aggregate. This approach is good if the input values in the real application change slowly. If they are volatile, the estimations of these protocols become more and more imprecise. The fact that an aggregation protocol needs a certain number of cycles to converge is an issue especially in larger or mobile sensor networks. One way to solve this problem is to increase the data rate of the network accordingly and to restart the protocols more often. If this is not feasible, because for example energy restrictions in a low-power sensor network application prohibit increasing the network traffic, dynamic aggregation protocols may help. They represent a sliding average of the approximated parameter and are able to cope with changing input data. In each protocol step, they will incorporate their old state, the received information, and the current input data into the calculations. A dynamic distributed average protocol like the one illustrated in Figure 6 is a weighted sum of the old estimate, the received estimate, and the current value. The weights in the sum can be determined by the Genetic Programming process

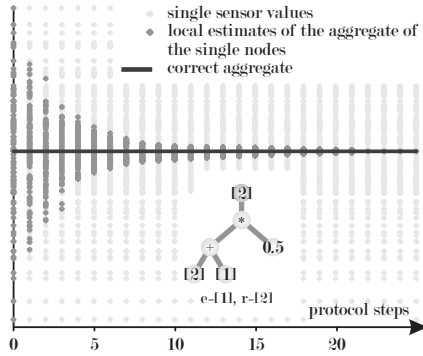


Fig. 5: The behavior of the distributed average protocol with static inputs

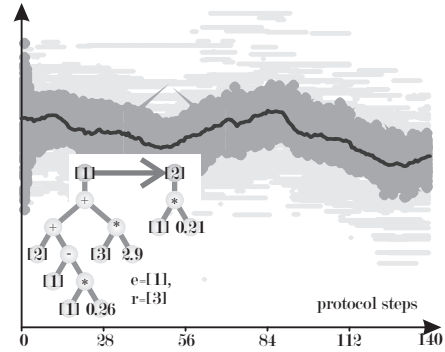


Fig. 6: A dynamic aggregation protocol for the distributed average

according to the speed with which the inputs change. In order to determine this speed for the simulations, a few real sample measurements would suffice to produce customized protocols for each application situation.

## 5 Results from Experiments

For our experiments, we have used a simple elitist evolutionary algorithm with a population size of 4096 and an archive size of 64. In the simulations, 16 virtual machines were running, each holding a state vector  $\mathbf{s}$  with five elements. For evaluation, we perform 22 simulation runs per protocol where each run is granted 28 cycles in the static and 300 cycles in the dynamic case. In the evolution, we put most of the pressure on optimizing the first objective, and only take the other values into consideration to break ties. This tiered comparison structure [11] leads to optimal sets with few members that most often (but not always) have equal objective values and only differ in their phenotypes.

### 5.1 Average – static

With this configuration, protocols for simple aggregates like minimum, maximum, and average can be obtained in just a few generations. We have used the distributed average protocol which computes  $\alpha_{avg} = \bar{x}$  in many of the previous examples, for instance in Section 2.2 and in Figure 4.

The evolution of a static version of such an algorithm is illustrated in Figure 7. It shows how the values of the first objective function (the mean square error sum) improve with the generations in twelve independent runs of the evolutionary algorithm. All runs actually converged to the optimal solution previously discussed, most of them very quickly in less than 50 generations.

Figure 8 reveals the inter-relation between the first and second objective function for two randomly picked runs. Most often, when the accurateness of

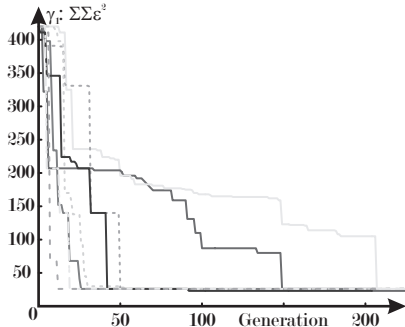


Fig. 7: The evolutionary progress of the *average* protocol

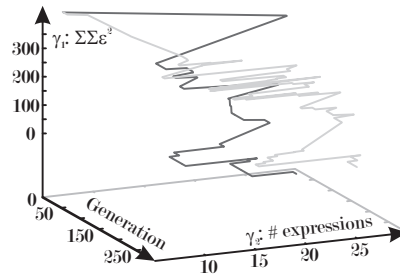


Fig. 8: The relation of  $\gamma_1$  and  $\gamma_2$  in the *average* protocol

the (best known) protocols increases, so does the number of formula expressions. These peaks in  $\gamma_2$  are always followed by a recession caused by stepwise improvement of the protocol efficiency by eliminating unnecessary expressions. This phenomenon is rooted in the tiered comparison that we chose: A larger but more precise protocol will always beat a smaller, less accurate one. If two protocols have equal precision, the smaller one will prevail.

## 5.2 Root-Of-Average – static

In [3] we used the evolution of the *root-of-average* protocol as benchmark problem. Here, a distributed protocol for the aggregate function  $\alpha_{ra}$  shall be evolved:

$$\alpha_{ra}(\mathbf{x}) = \sqrt{|\mathbf{x}|} \quad (11)$$

One result of these experiments has already been sketched in Figure 4. Figure 9 is a plot of eleven independent evolution runs. It also shows a solution found after only 84 generations in the quickest experiment. The values of the first objective function  $\gamma_1$ , denoting the mean square error, improve so quickly in all runs at the beginning that a logarithmic scale is needed to display them properly. This contrasts with the simple *average* protocol evolution where the measured fitness is approximately proportional to the number of generations. The reason is the underlying aggregate function which is more complicated and thus, harder to approximate. Therefore, the initial errors are much higher and even small changes in the protocols can lead to large gains in accurateness.

The example solution contains a useless initialization sequence. In the experiments, it paradoxically did not vanish during the later course of the evolution although the secondary (non-functional) objective function  $\gamma_2$  puts pressure into the direction of smaller protocols. For the inter-relation between the first and second objective function, the same observations can be made as in the *average* protocol. Improvements in  $\gamma_1$  often cause an increase in  $\gamma_2$  which is followed by an almost immediate decrease.

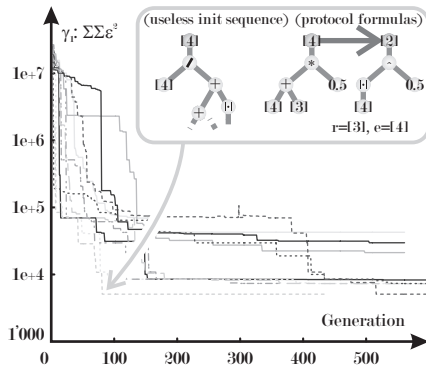


Fig. 9: The evolutionary progress and one grown solution of the *root-of-average* protocol.

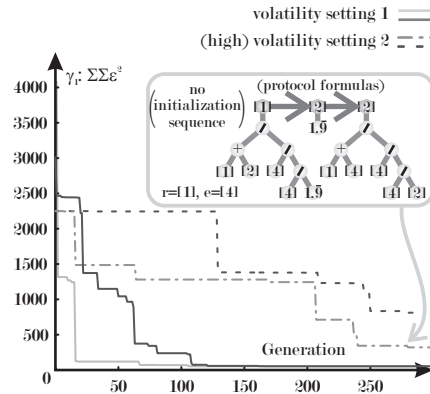


Fig. 10: The evolutionary progress of the dynamic *average* protocol.

### 5.3 Average – dynamic

Dynamically changing inputs are more interesting, since creating protocols for this scenario by hand is more complicated. We first repeat the “average” experiment for two different scenarios with volatile input data. The first one is depicted with solid lines in Figure 10. Here, the *true* values of the aggregate  $\alpha(\mathbf{x}(t))$  can vary in each protocol step by 1% and in one simulation by 50% in total. In the second scenario, denoted by dashed lines, these volatility measures are increased to 3% and 70% respectively.

The different settings have a clear impact on the results of the error functions – the more unsteady the protocol inputs, the higher will  $\gamma_1$  be, as Figure 10 clearly illustrates. The evolved solution exhibits very simple behavior: In each protocol step, a node first computes the average of its currently known value and the new sensor input. Then, it sets the new estimate to the average of this value and the value received from its partner node. Each node sends its current sensor value. This robust basic scheme seems to work fine in a volatile environment.

### 5.4 Root-Of-Average – dynamic

Here we follow the same approach as for the dynamic average protocol: Tests are run with the same two volatility settings as in Section 5.3. For the tests with data changing more slowly, we got a similar process of  $\gamma_1$  like in Figure 9. However, we found that the evolutions with the highly dynamic input dataset did not yield functional aggregation protocols. We suppose that there is a threshold of volatility from which on Genetic Programming is no longer able to breed stable formulas.

Again, in every experiment run, increasing  $\gamma_1$  is usually coupled to a deterioration of  $\gamma_2$  followed by a recreation span where the formulas are reduced in size. After a phase of rest, where the new protocol supposable spreads throughout the population, this cycle starts over again until the end of the evolution.

## 6 Conclusions

In this article we have illustrated how Genetic Programming can be utilized for the automated synthesis of aggregation protocols. The transition to the evolution of protocols for dynamically changing input data is a step towards a new direction. Especially in applications like large-scale sensor networks, it is very hard for a software engineer to decide which protocol configuration is best. With our evolutionary approach, different solutions could be evolved for different volatility settings which can then be selected by the network according to the current situation. The practical utilization of this new technique will be our next step in future work.

## References

1. T. Weise and K. Geihs. Genetic programming techniques for sensor networks. In *5. GI/ITG KuVS Fachgespräch "Drahtlose Sensornetze"*, 2006, pages 21–25, Stuttgart, Germany.
2. T. Weise and K. Geihs. Dgpf – an adaptable framework for distributed multi-objective search algorithms applied to the genetic programming of sensor networks. In *2nd International Conference on Bioinspired Optimization Methods and their Application, BIOMA 2006*, pages 157–166. Ljubljana, Slovenia.
3. T. Weise, K. Geihs, and P.A. Baer. Genetic programming for proactive aggregation protocols. In *Adaptive and Natural Computing Algorithms, 8th Internat. Conf., ICANNGA 2007*, Warsaw, Poland. vol. 4431 Springer LNCS, p. 167–173.
4. R. van Renesse. The importance of aggregation. In *Future Directions in Distributed Computing*, volume 2584 of LNCS, pages 87–92. Springer, 2003.
5. C.-Y. Chong and S.P. Kumar. Sensor networks: evolution, opportunities, and challenges. *Proceedings of the IEEE*, 91(8):1247–1256, 2003.
6. M. Jelasity, A. Montresor, and O. Babaoglu. Gossip-based aggregation in large dynamic networks. *ACM Trans. Comput. Syst.*, 23(3):219–252, 2005.
7. D. Kempe, A. Dobra, and J. Gehrke. Gossip-based computation of aggregate information. In *Proceedings of 44th Symposium on Foundations of Computer Science (FOCS 2003)*, 2003, pages 482–491, Cambridge, USA. IEEE Computer Society.
8. J.R. Koza. *Genetic Programming, On the Programming of Computers by Means of Natural Selection*. The MIT Press, ISBN: 0262111705, 1992.
9. X.H. Nguyen et al. Solving the symbolic regression problem with tree-adjunct grammar guided genetic programming: the comparative results. In *IEEE Congress on Evolutionary Computation, CEC2002*, 2002, pages 1326–1331, Honolulu, USA.
10. H.S. Lopes and W.R. Weinert. EGIPSY: an enhanced gene expression programming approach for symbolic regression problems. *Int. J. of Ap. Math. and Com. Sci.*, 14, 2004.
11. Thomas Weise. *Global Optimization Algorithms – Theory and Application*. 2007. See <http://www.it-weise.de/>.

```

@inproceedings{WZG2008DGPFAEPAP,
  title      = {Evolving Proactive Aggregation Protocols},
  author     = {Thomas Weise and Michael Zapf and Kurt Geihs},
  affiliation = {University of Kassel, Wilhelmsh{"o}her Allee 73,
                D-34121 Kassel, Germany},
  DOI       = {10.1007/978-3-540-78671-9_22},
  booktitle = {Genetic Programming -- Proceedings of the 11th European
                Conference on Genetic Programming, EuroGP 2008},
  location  = {Naples, Italy},
  month     = mar # {"26--28"},
  year      = {2008},
  pages     = {254--265},
  series    = {Lecture Notes in Computer Science (LNCS),
                Theoretical Computer Science and General Issues},
  volume    = {4971/2008},
  ISSN     = {0302-9743 (Print) 1611-3349 (Online)},
  ISBN     = {3540786708, 978-3-540-78670-2},
  editor    = {Michael O'Neill and Leonardo Vanneschi and
                Steven Gustafson and Anna Isabel {Espancia Alc{'a}zar} and
                Ivano{e} {De Falco} and Antonio {Della Cioppa} and
                Ernesto Tarantino},
  publisher = {Springer-Verlag GmbH, Berlin/Heidelberg, Germany},
  url      = {http://www.it-weise.de/documents/index.html\#WZG2008DGPFa},
  url      = {http://dx.doi.org/10.1007/978-3-540-78671-9_22},
}

```