2012 IEEE/SICE International Symposium on System Integration (SII)
Kyushu University, Fukuoka, Japan
December 16-18, 2012

# Publish/subscribe assisted neuroevolution in virtual life game

Mikko Polojärvi, Jukka Riekki and Masashi Furukawa

*Abstract*— **Simple Event Relaying Framework (SERF) is a software framework that attempts to bring lightweight publish/subscribe architecture inside individual devices and processes. In this paper we study the performance of SERF when it is used as a base for studying a different topic: virtual creatures in a life game learning to seek food via evolutionary algorithms. We measure the time spent on the whole evolution as compared to the time spent on routing events in different network configurations. Results indicate that SERF has small enough overhead to be used even for messaging within the same application while still enabling the benefits of loose coupling.**

## I. INTRODUCTION

Publish/subscribe (pub/sub) is a prominent modern messaging paradigm, where message senders do not explicitly define the receiver of the message being sent, or receivers the sender of the message being expected. Instead, messages are routed from senders (publishers) to receivers (subscribers) solely based which message topics or content the subscriber has expressed interest, usually with aid of an intermediating broker component. This allows for both parties to continue working with or without the presence of the other party. [1].

The primary advantage of pub/sub is that it allows loose coupling of the components: components can be easily added, removed or modified without breaking the system. Components do not need to know the location of other components, or even the time when a particular message was sent. This allows for more flexible network layout. [1].

Naturally, pub/sub also has its limitations. Components in pub/sub systems can be said to still be tightly coupled, if not to each other, but to the agreement on the semantics of message topics or contents in the given application. Therefore careful design in the early development stages is essential. Furthermore, many existing systems focus on components communicating through a high level computer network such as the Internet. Many such implementations have too large communication overhead for feasible utilization in lower level application architecture, such as the architecture within a single device or process.

Simple Event Relaying Framework (SERF) is a novel software framework attempting to bring pub/sub into low level software architecture by focusing on lightweight solutions in message routing and transmission. Although our earlier work [2] and [3] already presented a working framework prototype, more work is needed to analyze in more detail the advantages and disadvantages SERF presents in light of other software frameworks.

Our approach for this paper is to evaluate the framework when it is being used to solve a different problem on a different field where using SERF could offer clear advantages. One such application area is evolutionary algorithms (EA), which are characteristically complex and computation intensive. We hypothesize that low-level pub/sub can help divide the computational load to small pieces that are easy to parallelize.

As for the application to be created using SERF, we have chosen a tool for performing evolutionary computations in a virtual life game, where the environment is populated by numerous point-like creatures. One analogy would be imagining a view of the face of the Earth from high up in the sky, humans being just small dots that make decisions, do actions and interact with each other. Because one purpose of SERF is to facilitate spontaneous architecture modifications, we assume the tool will be used to calculate various different kinds of evolutionary tasks, involving different kinds of virtual creatures in different kind of test settings. This approach presents a challenge for parallelization, since it is not apparent where the performance bottleneck will form.

In this study we create a simple test featuring a single creature moving toward a single specified target, a setting often used to test EAs. This time the focus of our study is not on the actual test or performance of the EA, but instead on studying efficient SERF design methodology and SERF's performace running the simple EA. The study can be extended later to encompass more complex systems.

This paper is structured as follows. In the second chapter we examine other work related to the subject. The third chapter explains shortly how SERF works. The fourth chapter describes the architecture of our life game, followed by detailed explanation of the test settings, the used EA and the test results. In fifth chapter we discuss our findings and evaluate SERF on the basis of the tests. Finally, we conclude the paper.

## II. RELATED WORK

We are not aware of other similar solutions SERF could be easily compared to, because most pub/sub solutions do not consider bringing pub/sub inside individual processes. MQ Telemetry Transport (MQTT) protocol [4] bears some similarities with SERF in that it aims for enabling lightweight publish/subscribe, but it focuses primarily on minimizing the network traffic for resource constrained devices. Other solutions such as CLM [6] aim for lightweight pub/sub, but make definitions about used message format and transport

Mikko Polojärvi and Jukka Riekki are with Intelligent Systems Group and Infotech Oulu, University of Oulu, Finland {mikko.polojarvi, jukka.riekki}@ee.oulu.fi
Masashi Furukawa is with School of Information Science and Technology, Hokkaido University, Japan mack@complex.ist.hokudai.ac.jp

protocol. The Erlang programming language is an interesting comparison to serf, as it features lightweight processes with minimum size of only 309 words of memory [5].

There are numerous studies investigating properties of various evoluationary algorithms by applying them in a hypothetical problem area. One such widely used problem is the pole balancing (inverted pendulum) problem [9], a seemingly simple problem but which can be considered related to applications such as missile guidance systems. Virtual creatures seeking light sources is another classical test used to benchmark control systems.

## III. SIMPLE EVENT RELAYING FRAMEWORK

SERF is a software framework, i.e. a software library that offers a structure that implements some common functionality for a particular purpose. This structure can be extended to form a concrete software application. The main purpose of SERF is to facilitate implementation, modification and reuse of distributed applications composed of large number of software components. SERF attempts to do this by enabling lightweight pub/sub messaging for small software components communicating mostly within the same process.

### A. Events

In SERF, messages are called events. SERF does not concern itself with the actual contents of the event, instead handling it just as a blob of binary data. After publishing, the event is kept constant and unchanging. In case the event needs to be transmitted through network, the event can be serialized to binary format and back any time. Actual data structure or semantics of the contents are not defined by SERF, but instead left for application designers to decide. The general principle is that the events should primarily describe the situation from a single component's viewpoint. One of our research tasks is to investigate how to define the event contents effectively.

### B. Network structure

In technical sense, the primary purpose of SERF is to offer a method of deciding which subscribers each event is routed to. The event relaying network consists of three different kinds of components: nodes, links and processors. Processors are pieces of dynamically loaded executable program code, that can function as both publisher and subscriber. Processors are connected to nodes that function as a common thread execution environment for all processors connected to it. Furthermore, as a difference to earlier versions, the nodes also act as an application-specific data storage for the processors. In order to prevent race conditions, data stored in a node can only be accessed from processors attached to the node. Threads from common thread pool are assigned to work on nodes whenever they require attention.

In case the nodes reside within the same shared memory, they can connected together via two-way links, which are used to transfer ownership of events between nodes efficiently via shared memory reference passing. Nodes residing in different processes or devices can be linked together by adding a processor that transforms events in serial form, transmits the data packet using other means to a destination, where the event is reconstructed and published anew. Using TCP/IP sockets to transmit the packets is one possibility.

This way the events are relayed from processor to node, node to node and finally node to processor, possibly encountering a number of network transfers on the way. No central message brokers are used at any point. As a tradeoff for performance, the nodes do not track previously processed events, for which reason the event relaying network topology must be kept acyclic in order to prevent events from going endless circles in the system.

### C. Event filtering

The current version of SERF uses a simple filtering solution to prevent generated events from flooding the whole network, developed from the filtering solution described in [2]. Upon publishing, events are included with additional information called the event tag, which describes the contents of the events. Similarly, each node to node and node to processor link contains two event filters (one for each direction) that are updated automatically by the framework. Tags and filters are both represented logically by a binary image. Event filter allows an event to pass one way only if the event tag image is completely contained in the event filter. This is illustrated in Fig. 1. This way, the needs of one branch of the network are completely contained in one binary image. This allows the filtering decisions be made fast via binary operators, filtering out whole branches of the network with a single filter check. The underlying memory representation of the event tags and filters is application specific.

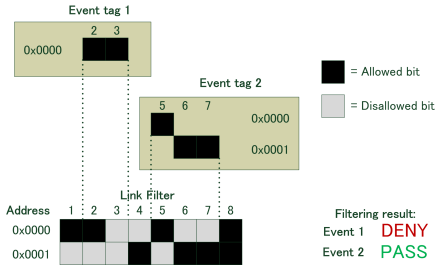


Fig. 1. SERF event tags and filters represented by binary images

SERF itself does not define any particular meaning for any of the bits in the binary image. Application designers are free to assign any meaning to any bit or combination of bits. Besides investigating effective ways to define the event contents, effective definition of the event filters is topic for further study.

### D. Advantages and limitations

The primary advantage of SERF is that it allows pub/sub messaging in low level settings were it is seldom used. Software components do not care where in the network other components reside, or which components produce/consume information. The result is that components can be effortlessly moved to different processes or machines without breaking

the system. Therefore there is less need to understand the nature of performance bottlenecks during the software development process. Second, because the framework encourages division of the software in subtasks with an established messaging scheme, the framework inherently promotes parallel computing and reduces synchronization problems. Third, SERF encourages exposing the internal state of the application for later utilization in new ways that were not planned during the application development phase.

Naturally, SERF has its limitations. Besides the limitation to purely acyclic (treelike) networks, event filter definition is another factor limiting the scalability of the framework, because altering the definition may require recompilation of the whole application. However, SERF was never intended as an universal networking solution. It is entirely feasible to use SERF to implement parts of the application where it works best, leaving the rest for other methods.

### E. Evaluation

Evaluating the performance of SERF depends greatly on the application area and the specific way SERF is utilized in the application design. In this study, in the context of our virtual life game, we attempt to evaluate the performance of the framework with the following measurements:

1) Number and volume of events sent by publishers
2) Number and volume of events received by subscribers
3) Total number and memory usage of link filters
4) Time spent doing filtering checks vs total running time
5) Effect of the number of threads and amount of concurrent computation on execution time.

## IV. LIFE GAME

As mentioned earlier, the object of our study in this paper is a tool for studying the behavior of evolved point-like virtual creatures in 2D space. The creatures move around in the virtual world, make decisions, do actions and interact with each other. In this study we focus on the strategic decisions made by the creatures: where to walk, where to look, what to eat etc. During the development of the tool, we do not yet know exactly what capabilities our creatures will have, how many creatures there will be or what kind of tasks facing or even what EA we will be using in the future. Different intermediate findings may take our study to completely different directions. We hypothesize this kind of problem setting is exactly where utilizing SERF can offer advantages.

### A. Entity structure

As we do not know exactly what kind of creatures we will be working on later, we want to make as few assumptions about the capabilities of the creature as possible. First, we introduce the concept of entity. An entity may be any single object in the environment such as a controllable virtual creature or an immobile food object, or a bigger group of simpler creatures that are represented with the same logic. Second, we establish that all entities are composed of a number of *features* of which each represents a single facet of the entity. Each feature is furthermore divided into two parts: the current state of the feature and the logic that makes the feature perform its function. The feature states are stored in a SERF node, while the logic part is represented with a SERF processor.

TABLE I
CREATURE FEATURES SUMMARIZED

| Feature | I | O | State | Logic |
|---|---|---|---|---|
| Body | | | location<br>orientation | on **bites at X** and near X:<br>send **I was eaten**<br>on **looking at X**<br>send **I was seen**<br>on **body pushed**: move<br>on **body rotated**: turn |
| Life | | | energy<br>age | each round: reduce energy<br>each round: increase age<br>when out of energy: die |
| Legs | 1<br>1 | | velocity<br>rotation | on **input**: send **body pushed**<br>on **input**: send **body rotated** |
| Eyes | | 1<br>1<br>1 | food in sight<br>food distance<br>food bearing | each round:<br>send **looking at X**<br>on **I was seen**: send **output** |
| Mouth | 1 | | eating | on input: send **bites at X**<br>on **I was eaten**: add energy |

As for our first test, we implement a simple creature entity with five features: body, life, legs, eyes and mouth. We also implement a food entity that has just the body feature. The roles of each feature is summarized in table I. Event types are in **bold**, and cause and effect in logic are separated with a colon ':'. The I and O columns show whether or not the state variable on the right is exposed in **input** and **output** events, respectively. The features communicate with other features in other creatures by publishing and subscribing to events. For example, when a creature uses its eyes to look around, it sends a **looking at X** event with description of the location being looked at. The body feature in each other creature determines if the body is within the location being looked at, in which case it replies with a **I was seen** event. Upon receiving that event, the eyes publish sensor **output** events, which external creature controller processors may then use to make decisions and supply features such as legs actuator **input** events.

The nodes and processors all entities in the environment arranged in a tree-like network as depicted in Fig. 2. Each entity node (left) is connected to a entity class node (middle) which contains just one processor that is responsible for generating more nodes when entities get born in the life game. Entity class nodes are bound together by a single root node (right).

For example, $n$ being the number of other entity classes and $m$ the total number of visible entities in all other classes, in order to get a **looking at X** event from eye processor to the body processor of all other entities, the event needs to travel the following path: eye processor $\rightarrow$ entity node $\rightarrow$ entity class node $\rightarrow$ entity class root node $\rightarrow$ each other entity class node $\rightarrow$ each entity node $\rightarrow$ each body processor. Thus, $3 + n + 2m$ passed filtering checks are required. For each

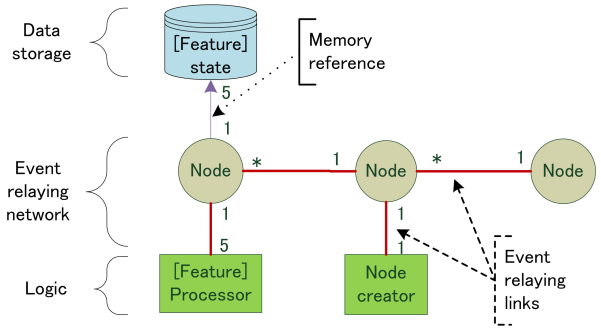


Fig. 2. SERF event relaying network formed by creatures and their feature processors

seen entity, another 6 passed filtering checks are required to get an **I was seen** event back to the eye processor of first entity. Additionally, for each fork along the path, a number of denied filtering checks are also required to ensure the events reache only their corresponding subscribers.

### B. Event filters

Fig. 3 describes the simple scheme we used to divide the event filter bitmap into five different parts. First, in order to be able to designate a particular feature in a particular entity and entity class using just the event filter, we assign one bit from the event filter bitmap for each. Additionally, because we will run several parallel life games simultaneously, we designate a number of bits to signify the "plane" where the life game occurs. Finally, we dedicate one bit per possible event type that can occur during the simulation.

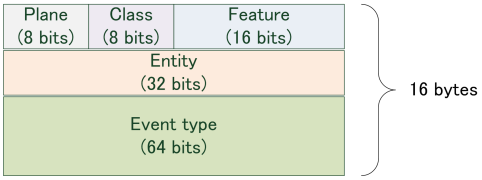


Fig. 3. SERF event filter bitmap with semantic partitioning

### C. Event content

In order to reduce the processing, serializing and deserializing computations to minimum, the following method is used in our life game. In addition to being used to route the events between publishers and subscribers, the event type bit in the event filter is used to also determine the data structure of the event. SERF processors simply checks the event type bit and casts the event content blob to a corresponding data structure. For example, if the legs processor in an entity is sent an **input** event, the processor interprets the event content as two float variables and uses them to move the legs.

### D. Flow of the game

The life game is divided into rounds, with each round divided to four phases: upkeep, input, action and output. A dedicated SERF processor *scheduler* announces the start of a new phase with an event and waits for any other processors to react. When all triggered processors have finished processing, the scheduler instantly announces the next phase. In the "upkeep" phase, creatures age and lose energy. In the "input" phase entities controlling the creatures send input to creature actuators (such as legs). In the "action" phase features perform their function according the received input (such as moving or looking around). In the "output" phase the features publish sensor output to entity controllers. Each event tag generated during the game in a particular plane has its corresponding plane bit enabled, to prevent the events from leaking to other planes. Because the events controlling the internal state of the game can be freely subscribed to like any other, this structure makes it possible to monitor the internal state of the game by just connecting a processor in the network and subscribing to desired data, without altering any of the components involved.

### E. Flow of the evolution

In the following description, SERF processor names are printed in *cursive* for clarity. At first, the whole SERF network consists of only a few processors: one *keyboard input monitor*, one *evolver loader*, one *network monitor* and a number of *evaluator loaders*. Upon starting the evolution, the *evolver loader* extends the network with a *test environment creator* and an *EA*, both loaded from a software library. The *EA* creates an initial population of genomes and instructs the *evaluator loaders* to extend the network with *evaluators*, each being assigned to a separate plane. When each *evaluator* announces its readiness, the *EA* requests it to evaluate one particular genome. When each genome has been evaluated, the *EA* evolves the generation and resumes assigning genomes for evaluation until the algorithm finishes.

Upon receiving an evaluation request, each *evaluator* requests the creation of a test environment as described in Fig. 2. The *evaluator* creates an *creature controller*, giving it an artificial neural network (ANN) created according to the genome being given for evaluation. Finally, the *evaluator* creates a *scheduler* and tells it to start the simulation. The *creature controller* listens to "input phase" and "sensor output" events generated by the *scheduler* and the *features*. The ANN is used to convert received sensor output events into actuator input events that in turn control the creature in the test environment, according to Fig. 4.

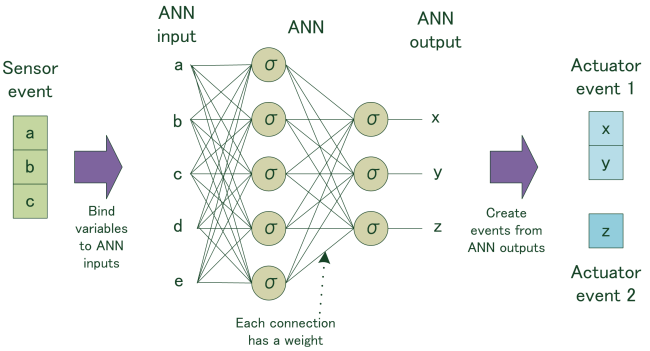


Fig. 4. ANN generating actuator input events from sensor output events

While the simulation runs, the *evaluator* listens to events generated by the creatures in the test environment and uses the information to evaluate the genome's performace. When the evaluation finishes, the *evaluator* publishes the score attained by the genome, destroys the test environment and returns to idle state.

During any time, the *network monitor* may be used to connect other SERF networks in the system. Since all events are transmitted through the socket as normally according to subscriptions, it can be used to monitor, visualize or administer the system even from a different terminal.

### F. Test setting and evolutionary algorithm

In our first test we place a single creature entity a distance away from an immobile food entity, initially facing away from it. The food entity is exactly the same as the creature, except that it lacks all features and processors except the body. Basically we want the creature to evolve to use its eyes to turn to face the food object, move toward it with legs and finally consume the food with mouth, staying close to the food long enough to finish the task. The test is repeated a number of times, each time facing the creature to a different initial direction in order to prevent the algorithm from dumbly learning to turn and move a fixed distance, disregarding the sensors. The final fitness is equal to the average of all attempts. For a single attempt, the fitness $F$ is

$$F = L + c_s S + c_d D + c_o O,$$

where $L$ is the creatures lifetime in rounds, $S$ the number of rounds with no food in sight, $D$ the distance to the closest food in sight and $O$ the angle between the creatures orientation and the vector from the creature toward the food. $c_s$, $c_d$ and $c_o$ are coefficients adjusting the importance of each measurement.

As we expect that during our studies changing test settings will cause optimal ANN topologies to change as well, it is logical to look for EA solutions that are able to evolve the ANN topologies as well as connection weights. For our study, used a slightly modified version of Feature Selective Neuroevolution of Alternating Topologies (FS-NEAT) algorithm [7]. FS-NEAT differs from regular NEAT algorithm [8] in that it starts with only minimal set of initial connections instead of a fully connected ANN, using evolution to decide which inputs and outputs are relevant for solving the problem.

### G. Test results

The measurements benchmarks were executed on a single PC with Intel Core i5-2500K CPU featuring four processor cores running at 3.30GHz and 8 GB memory. The operating system was 32-bit Ubuntu 12.04, running kernel version 3.2.0-29. All software including the current SERF prototype were written with C++.

In 8 times out of 10, the used EA succeeded in evolving a creature that is able to locate and consume the food entity before dying within 200 generations. In each success the EA produced an ANN that had connected the necessary sensor inputs to proper actuator outputs, whereas in each failure the EA failed to "find" the correct connections.

TABLE II
SERF BENCHMARK RESULTS

| Threads | | P/S | Total number of evaluators 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|---|
| Execution time (s) | 1 | P | 118 | 58.2 | 39.8 | 34.8 | 35.8 |
| | 1 | S | 17.4 | 22.8 | 23.6 | 25.8 | 26.5 |
| | 2 | S | 21.3 | 18.0 | 15.3 | 17.9 | 18.8 |
| | 3 | S | 24.7 | 14.9 | 16.5 | 17.3 | 16.2 |
| | 4 | S | 27.5 | 16.5 | 15.2 | 17.1 | 18.4 |
| Tot. number of links | | | 60 | 82 | 104 | 126 | 148 |
| Tot. link filter size (B) | | | 1308 | 1812 | 2316 | 2820 | 3324 |
| Passed filter checks ($10^6$) | | | 8.65 | 9.95 | 10.3 | 11.2 | 10.8 |
| Denied filter checks ($10^6$) | | | 16.7 | 22.0 | 24.0 | 27.6 | 28.0 |

The performance of SERF was measured first measuring the execution time required for evaluating total of 256 creatures within 10 generations with varying number of worker threads and concurrent evaluation planes (ev) residing in shared memory (S) or separate processes (P). Each simulation consisted of 200 rounds that were repeated 4 times for each genome. The results are shown in Table II, with Fig. 5 describing the shared memory measurements.

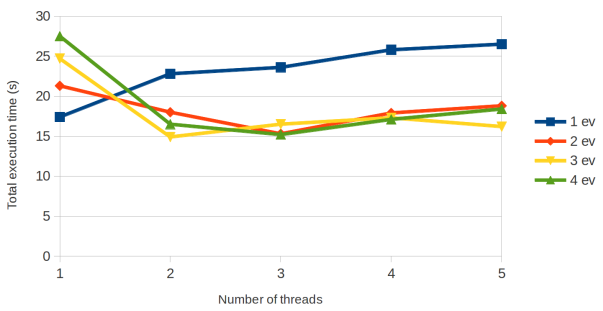


Fig. 5. Execution time for 10 generations, shared memory

In addition, during the execution 1.3-1.5M published events are received 2.7-3.1M times in various SERF processors, each event on average containing 20 bytes of data and 20 bytes defining the event tag. Total time spent handling 11.3M passed filtering checks was measured 301 ms, while denied filtering checks required 27 ms. On average, a single passed filtering check took 27 ns and a denied one less than 1 ns. This can be compared to a 20 byte character string comparison that on the test equipment took around 10 ns. All measurements stayed relatively constant from run to run, and the number of generations or the performance of the simulated creatures did not affect the results considerably.

## V. DISCUSSION

### A. Life game evaluation

The life game and the test setting described in this paper were kept intentionally simple. Although the described life game does not generate especially large number of events per round of simulation, one simulation typically contains a large

number of rounds. Furthermore, the evolutionary algorithm necessitates repeating the simulation once for each attempt, genome and generation, ending up with a large total number of published events (1.3-1.5M). Although the setting does not yet model any particular real life problem, it highlights the need of handling the event transmission effectively, making it a useful test application for studying the performance and usage of SERF. Furthermore, it serves as a stepping stone for studies involving more complex problems. We identify two main requirements for real life problems where EAs and the life game simulation built on SERF could be used effectively: the simulation environment needs to model reality sufficiently accurately, and the resulting dynamics should be complex enough to warrant looking for the solution with EAs in the first place.

### B. EA evaluation

All in all, the FS-NEAT algorithm manages to evolve a food seeking creature without problems. The biggest challenge for the EA turned out to be finding a suitable ANN topology for the task at hand, or more specifically, finding and connecting the necessary inputs to correct outputs. Without using the correct inputs the evolution would stall for several generations until the EA finds a beneficial topology mutation, whereupon the fitness of the creatures starts proceeding rapidly.

### C. SERF evaluation

As shown in Table II, evolving 10 generations on the test equipment required 14.9-27.5s. Compared to that, computing the 11.3M passed filtering checks required significantly less time 301ms, with the denied checks getting denied even faster, in 27ms. This supports our claim that the SERF network can be expanded without overburdening the whole network, given that processors communicating often with each other can be kept close together. In the life game most event traffic happened within individual evaluator planes. For events that concerned only one particular plane, unrelated branches of the network could be removed from routing with small number of filtering checks.

Table II also shows that best performance is achieved by having a suitable balance between worker threads and amount of nodes requiring attention simultaneously. With too many threads they often end up waiting for each other when encountering critical sections. With too many evaluation planes, the extra evaluation planes do not get worked enough to provide increased performance. When the evaluators were placed in separate processes, the execution speed was considerably slower due to necessitating serialization and deserialization of the events. However, in this case the effect of adding evaluators was more pronounced, with the total exceution time for a single evaluator (118s) getting rougly divided by the number of evaluators, up to the limit of four processor cores.

At all times during the evolution, each processor was kept ignorant of the location of other processors in the network. This made it possible to freely divide the nodes and processors to different processes late in the development process. In case more computational resources are needed, some parts of the SERF network can be simply moved to other machines, even when the location of the performance bottleneck was not known in the design phase, though the resource boost must still beat the overhead resulting from serialization and deserialization. Although more studies are required to understand the effects of different network structures to more detail, the observations thus far support the claim that the pub/sub application design method promotes parallelization by facilitating division of computational load.

### D. Future work

The next step in our work is to continue studying SERF and the life game in more complex test settings involving multiple more complex creatures. Furthermore, we seek to model and study a real life problem with the life game presented in this paper.

## VI. CONCLUSIONS

In this paper we described the current version of SERF, a simple software framework aiming to enable pub/sub messaging in low level settings. We used SERF to implement a life game simulation engine that models point-like creatures in 2D space. In order to provide a computation intensive setting, we used the engine to solve an evolutionary problem that aims to evolve a creature that is able to locate and consume food using available sensors and actuators. During the test SERF is able to handle the large number of events generated by the simulation engine without getting congested and finish the evolution in short time. At any time components of the simulation engine could be freely arranged in the pub/sub network without breaking the simulation, supporting our claim that SERF can offer the benefits of pub/sub even in low level settings.

## REFERENCES


[1] Eugster P., Felber A., Guerraoui R., Kermarrec A., "The many faces of publish/subscribe", *ACM Computing Surveys (CSUR)*, June 2003, vol. 35, issue 2, pp. 114-131

[2] Polojärvi M. and Riekki J., "Experiences in Lightweight Event Relaying Framework Design", *5th International Conference on Future Information Technology (FutureTech)*, May 2010, Busan, South Korea.

[3] Polojärvi M. and Riekki J., "Lightweight Service-Based Software Architecture", *Proceedings of the 6th international conference on Grid and Pervasive Computing (GPC'11)*, May 2011, Oulu, Finland, vol. 7096, pp. 172-179

[4] MQTT: MQ Telemetry Transport, `http://mqtt.org/`

[5] Ericsson AB, "Erlang efficiency guide", `http://www.erlang.org/doc/efficiency_guide/processes.html`

[6] Kusznir J., Cook D., "Designing Lightweight Software Architectures for Smart Environments" *6th International Conference on Intelligent Environments (IE)*, July 2010, pp. 220-224

[7] Whiteson S., Stone P., Stanley K., Miikkulainen R., Kohl N., "Automatic feature selection in neuroevolution", *GECCO '05 Proceedings of the 2005 conference on Genetic and evolutionary computation*, June 2005, pp. 1225-1232

[8] Stanley K., "Efficient Evolution of Neural Networks Through Complexification", PhD Dissertation, 2004, Department of Computer Sciences, University of Texas at Austin

[9] Geva S., "A cartpole experiment benchmark for trainable controllers", *Control Systems, IEEE*, vol. 13, issue 5, pp 40-51, Oct 1993