Self Evolving Services

Eero Kaukonen

Abstract— **Genetic algorithms and swarm intelligence are artificial intelligence techniques that are inspired by the way living things evolve and co- operate. Genetic algorithms aim to solve problems by breeding better solutions from a parent set of candidate solutions. Swarm intelligence aims to emerge intelligence from a swarm of simple agents. There are some experimental frameworks for self- evolving network services that utilize the ideas of biologically inspired artificial intelligence methods. In these frameworks services are provided by autonomous agents that can adapt, evolve and collaborate with other agents**

Index Terms— **network services, genetic algorithms, swarm intelligence, self-healing**

I. INTRODUCTION

Computing services have become more and more distributed and decentralized during the last couple of decades. Most of the everyday services used by ordinary consumers are available in the Internet. For example, nowadays there are very few people that do not reserve airline tickets or buy books online.

Often the network of clients and servers that is visible to the user is only a façade. The most successful web stores, such as Amazon and Dell, do not have things like traditional inventories at all. Their key to success has been in implementing a virtual supply chain that is in essence a globally distributed information system for handling orders and deliveries.

The increased importance and usage of networked services has posed several new challenges. In the old mainframe systems, the amount of simultaneous users and their usage patterns were steady and known in advance. Internet has changed most of this. Nowadays users can quickly move from using one service to another and usage patterns can change rapidly as fads come and go. For example, a few years ago Internet poker was just invented and only few players played online. Today the amount of online poker players is measured in hundreds of thousands, in Finland only.

It is also important to remember that even though software development today is easier than in 1960's, the software deployed on network servers has become increasingly complex. Most of development is done on some kind of framework, such as J2EE. When most of the low level operations are hidden behind the framework, it is very easy to dramatically change the way the application consumes resources. Opening a network socket or storing an object in a cache can be very resource intensive, but in most of the modern frameworks these happen automatically.

In this article I discuss how networked services can be modeled as biological entities that learn, evolve and adapt to their environment. In the second chapter I discuss genetic algorithms and swarm intelligence, biologically inspired

problem-solving methods. In the second chapter I present an experimental framework for evolving network services. In the fourth chapter I discuss some open issues and directions for future research.

II. Biologically inspired problem solving

A. Genetic Algorithms

Genetic algorithms [3] are a family of methods inspired by biology. Generally they work by breeding better solutions to a problem from a pool of inferior candidate solutions. They have been successfully applied to various optimization and constraint satisfaction problems. One classic example of a classic constraint satisfaction problem that can be solved is the knapsack problem, where various objects of different sizes must be packed into the knapsack while wasting as much space as possible. In real life the knapsack could be a sea container and the objects could be different cargo items that are packed into the container.

The standard operators used in genetic algorithms resemble the natural behavior of genes in living things. Most important part of a genetic algorithm is the fitness function. It gives each member of the population a fitness value that measures the 'goodness' of the individual solution. Only the fittest candidates are allowed to reproduce. A genetic algorithm thus aims to maximize the fitness of the solution.

Producing a new set of potential solutions from the fittest candidates from the previous round can be done by recombining the genes of two parent solutions into a new child solution. In this process random mutations may happen. A generic genetic algorithm can be presented in pseudo-code as follows:

```
geneticAlgorithm
{
    t = 0;
    create initial population P(t);
    evaluate P(t);
    until (done)
    {
      t = t + 1;
      select fittest from P(t-1) to
      P(t);
      recombine P(t);
      mutate P(t);
      evaluate P(t);
    }
}
```

One good example of a problem that can be solved by genetic algorithms is the classic traveling salesman problem (TSP), where a salesman has to visit n cities while minimizing the distance traveled. The routes the salesman can choose are limited by the availability of roads between the cities.

In a genetic algorithm for TSP, the genes are different routes and fitness function is the distance traveled. Connecting sequences from old genes can create new sets of genes. For example, one parent route could be (A, B, C, D, E, F) and another parent route could be (C, A, B, D, F, E). These two could be recombined two produce a new child route (A, B, C, D, F, E). If (A, B, C, D) is shorter than (C, A, B, D) and (F, E) is shorter than (E, F), the child route is better than either of the parent routes.

It is easy to see that if the new generation is created from the shortest routes of the old generation, the average length of the routes should converge towards the optimal solution. However, it should be noted that genetic algorithms are not guaranteed to yield optimal results. Usually the fitness function and the algorithm itself have several parameters that have to be fine tuned to get good enough results. As in real life evolution, accidence plays a big part in genetic algorithms. It is not always the fittest that survive.

B. Swarm intelligence

Swarm intelligence is one of the latest and hottest research topics in artificial intelligence [4]. It is a well-known fact, that individual insects such as bees and ants are not very smart. However, together they manage to solve complex problems such as building a nest or finding the shortest route to food.

A classic example of swarm intelligence is the way some ant species find the shortest route to food sources. Individual ants start their search for food in a pretty random fashion and their ability to figure out the shortest route is naturally quite limited. However, they leave traces incense that other ants can smell behind them when they travel towards the food source. Even though the ants do not know the concept of a map, they can follow the traces left by other ants. As the ants generally tend to follow the strongest smell and as the sell of the traces gets weaker as time passes by, eventually the route they select will be closer and closer to the optimal route.

The general idea of the ant algorithm for route finding is called ant colony optimization [4]. It has been successfully applied to practical problems such as packet routing in a network that does not have a static structure. There have also been some promising experiments with swarms of simple robots. For example, spider-like robots walk and climb much better if each leg is controlled by a simple, autonomous agent than if all the legs are controlled in a centralized manner.

C. Organic Network Services Today

It is easy to find examples of swarm like behavior in real life network services. Grid computing is an obvious example, but also services that are deployed on the same physical machine are often modeled as independent logical servers that collaborate to get the assigned tasks done. For example, J2EE servers are usually divided to different layers, such as static web, dynamic web, application and database layers. These layers usually consist of multiple logical or physical servers to achieve better fault tolerance and easier optimization.

Most attempts to evolve executable code with genetic algorithms have not yielded useful results. However, on an average application server there are plenty of configuration parameters that can have an enormous effect on performance, for example:

- The JVM memory management configuration. If garbage collection and heap sizes are incorrectly configured, very large amounts of processor time might be wasted.
- Execution threads. If there is not enough of them, processor time and memory might go to waste but if there is too many of them, high load may crash the server.
- Connections to database and other external resources. If there is not enough connections external computing resources available, execution threads might spend long times waiting for them.

Usually these parameters have to be hand tuned. As predicting and simulating actual user behavior can be tricky, tuning usually happens by trial and error. If services have tens of thousands concurrent users, errors can become very costly.

It is clear that there are things in modern network services that resemble swarms or genes. The biggest problem is that the intelligence of the swarms is hard coded in the network topology and system administrators drive the evolution of the genes by manually tuning them.

III. A Framework for Evolving network services

A. Architecture

An experimental framework for self- evolving services built by Nakano and Suda [1] can be divided into three different components [Picture 1].

- **Platform** provides resources required for performing services, such as memory, CPU time and network bandwidth. These resources are collectively known as **energy**.
- **Agents** are the entities performing the tasks. They have to pay in energy for the computing resources they use. The energy- usage of agents is tracked and those that consume all of their energy are eliminated by the platform.
- **Users** issue tasks for agents. They have to pay in energy for the services provided by the agents.

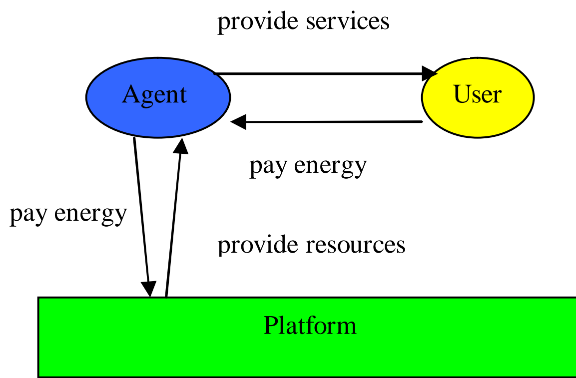

Figure 1: Framework for adaptive services

The individual agents do not have complete information about the whole system. They are only aware of local conditions, such as:

- **Request rate**: the amount of service requests in a time period
- **Request rate change:** how much the request rate changes
- **Population**: how many agents there are on the platform the agent is located
- **Resource cost:** how much different resources cost in terms of energy
- **Behavior cost**: how much different behaviors, such as reproduction, cost in terms of energy.

The framework is a sandbox where the agents can provide services, collaborate with each other and evolve. Nakano and Suda [1] did not implement the

B. Simulated Evolution

The agent's primary task is providing users with the service they require. During their lifecycle they can perform other tasks too, such as reproduction or migration to a new platform [1]. They can also delegate the service request to another agent, if they have enough energy. Agents that are using energy in an inefficient way will eventually run out of it and be eliminated by the platform. Natural selection is also applied in the reproduction operation. The most energy effective agents tend to have more offspring, as they are more highly valued in the partner selection.

The behavior invocation mechanism is similar to the method used in artificial neural networks. In neural networks, a synopsis will send a signal forward, if the sum of incoming signals is greater than a certain threshold value. In this framework, the incoming signals are the observations (request rate etc.) weighted by a set of values w_i [Figure 2].

The weights w_i are the genes of the agent. In essence, they

determine how energy-efficient an agent can be under the prevailing conditions. In reproduction, the genes of two agents are combined and a new agent is created from these genes. The new agent will continue to perform the same task its parents performed, and its behavior will share some features with both of its parents. It is easy to see how natural selection should eliminate inefficient behavior patterns. The agents should also be able to adapt to new conditions by reproducing offspring with slightly different behavioral patterns.

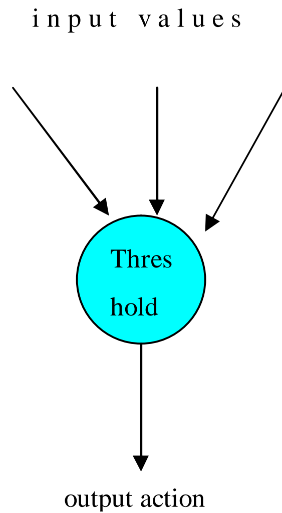

Figure 2: Behavior execution mechanism

C. *Results*

Nakano and Suda [1] evaluated their framework by running a simulation in a grid consisting of 64 virtual nodes, each one of them representing a virtual platform. Users could issue tasks to any of the platforms that had living agents. Their experiments also included special scenarios, such as failing platforms, platforms with different resource costs and varying workload. They found that evolving agents had better performance than the control group of non-evolving agents.

- On platforms with *varying resource costs*, agents placed on expensive platforms starved. Evolving agents were able to migrate out of the expensive platforms to cheaper ones.
- Evolutionary agents responded better to *increasing workload*. On non-evolving platforms the response time increased when the load increased, whereas on evolving platforms the agents reproduced more rapidly and were able to respond to increasing load.
- In case of *platform failures*, agents that did not have any evolutionary mechanisms quickly became extinct. Evolutionary agents were able to reproduce and eventually migrate back to the failed platform when it became available again.

IV. Future Directions

A. *Mobile Web Services*

Web services are a WWW based implementation of **service-oriented architecture**. In addition to the two parties of traditional client-server architecture, service oriented architecture has a third party called service registry. Clients use the service registry to look up service providers (servers). The service provider decouples the client from the server, making the operation of switching to another service provider easier. This is essential for adaptive systems. If the service provider and the client are tightly coupled to each other, it is not possible to switch to another service provider in case of network failures or other unexpected incidents.

Web services enable the composition of complex tasks from simpler, atomic services [5]. For example, holiday trip planning web service could be composed of flight booking, hotel booking and car rental services.

Orchestration of web services refers to decentralized execution of a composite task. In web service orchestration, there is no single process that executes the workflow. Individual services should be able to coordinate their execution themselves. If a mobile client had to coordinate the execution of a complex workflow of remote services, bandwidth and connectivity issues could be quite serious [5].

Sheng et. al. [5] are proposing a framework for enabling adaptive composition of web services, especially for mobile devices. They aim to solve the following issues:

- *Need for personalized composition.* As opposed to traditional services, mobile users are more sensitive to time and space constraints. For example, mobile pizza ordering service should be able to send orders to the nearest restaurant, even though the service user did not know which one it is.
- *Limited resources and wide variety of different handsets.* The computing resources on a handset are very limited compared to traditional client computers. Different devices also have different abilities to show graphics or animations and play sounds. This makes service customization even more important.
- *Robustness.* There are many situations in a mobile environment where the execution of a service could be interrupted. Also different quality attributes, such as response time, can vary dramatically. Adaptation is more important for mobile networks than it is for traditional services.

In the mobile framework designed by Sheng et. al. the key idea is to separate the process skeleton from the individual services. The process template is modeled as state transition diagram. As in ant colony optimization the responsibility for executing the process is distributed across the different agents and services.

B. Task awareness

The framework proposed Nakano and Suda was relatively simple. In real life problems there are many different services and different users with different needs. Optimizing only one quality attribute of the service, such as response time, is often not enough. For example, if user is downloading a movie she is going watch later, the response time is not that important and the user may prefer a better quality format that takes a bit longer to download. However, if the movie is in a streaming format and the user wants to watch it immediately, it might be preferable to decrease the quality of the stream to increase the response time of the service. In many real life services there are several, conflicting quality attributes.

Aura framework [2] was developed to address the issue of varying user preferences. Its goal is to make services aware f the task they are performing, so that they know which quality attributes are more important than the others. The Aura framework consists of three different layers.

Environment is the lowest layer of the framework. It provides the basic resources for executing tasks. The resources are traditional network services that can be configured by the upper layers. It also monitors the quality of service values.

Environment management layer is responsible for configuring the environment layer. It monitors user needs and environment resources and maps the tasks issued by users to the services provided by the platform. It is also responsible for finding new service providers if the old ones disappear.

Task management layer is responsible for monitoring user behavior. It decomposes complex tasks so that they can be executed by the lower level services. It is also responsible for tracking the user context and storing partially executed tasks if their execution is interrupted.

The key context of Aura framework is the utility function. It maps the user's context and preferences to a utility value that tells what is best for a certain user in a certain situation. The framework aims to maximize the utility function for different users in different situations.

Garlan et. al. [2] present mobile translation as an example of an service that would benefit from self adaptation. The mobile translation service consists of three separate services. Speech recognition service translates speech to text, translation service translates text in one language into text in other language and speech-synthesizing service reads the produced text in correct language.

Initially the whole translation process might run in the users' mobile phone. As the mobile phones computing resources are limited, but the cost of transmitting text data is very low, the translation service might be set up to run on a remote server. But if the remote server becomes unavailable or the connection gets severely weaker, there is a risk that the quality of the service drops into an unacceptable level. Now the environment management layer might make the decision to perform the translation on users handheld or switch to an alternative server, if there is one available.

V. Conclusions

In this article I discussed the idea of biologically inspired problem solving, especially swarm intelligence and genetic algorithms. As they model the aspects that make living things intelligent, evolving and adaptive, it seems natural that they could help us to build more robust, survivable and adaptable network services.

There are some experimental frameworks that utilize the biologically inspired AI methods. The results show that some evolution is better than no evolution, but the experimental frameworks are far away from practical, real life applications. They resemble the toy problems and solutions that made the early AI methods so popular in the late 70s and early 80s.

As discussed in chapter IV, real life self- adaptation is much more complex than sandbox simulations. There are real practical issues in communicating users' quality requirements to the computer or taking various changing environmental attributes into account.

A self-adaptive system that would have any practical value must consist of multiple layers. The composition of complex workflows and their execution must be clearly separated from each other in order to achieve loose coupling between service providers and clients.

A language for expressing and evaluating user requirements must be formally defined so that users and intelligent agents can communicate with each other. The service descriptions must have some machine-readable meaning so that autonomous agent can decide which services can be used to satisfy user requirements. This issue has been discussed in the context of semantic web, and it is definately not trivial.

As presented in this article, it is possible to simulate evolution and swarm intelligence, as well as execute complex workflows in a distributed manner. The real practical challenge is putting these two things together to provide a self-adaptive service that could be used to solve real- life problems.

References

[1] T. Nakano and T. Suda: “Adaptive and Evolvable Network Services”, GECCO-2004,

[2] D. Garlan, V. Poladian, B. Schmeri, J. Sousa: “Task-based Self-adaption” WOSS, pp. 54-57, ACM, 2004.

[3] K. de Jong: “Learning with Genetic Algorithms: An Overview”, Machine Learning, Vol. 3, p. 121, 1988.

[4] P. Tarasewich and P. McMullen: “Swarm Intelligence: Power in Numbers”, Commun. ACM, 45(8), pp. 62-67, 2002.

[5] Q. Sheng, B. Benetallah, Z. Maamar, M. Dumas, A. Ngu: ”Enabling Personaized Composition and Adaptive Provisioning of Web Services”