Handling Complexity in Some Typical Problems of Distributed Systems by Using Self-organizing Principles

Vesna Šešum-Čavić

Abstract Today's software systems are continuously becoming more complex. Main factors that determine software complexity are huge amounts of distributed components, heterogeneity, problem size and dynamic changes of the environment. These challenges are especially emphasized in distributed software systems. To cope with unforeseen dynamics in the environment and vast number of unpredictable dependencies on participating components, employing of self-organization principles at different levels in the software architecture can be beneficial. This could help in shifting complexity from one central coordinator component to many distributed, autonomously acting software components. Swarm intelligence represents a self-organizing biological system. Therefore, swarm-inspired algorithms play an important role in the design of self-organizing software for distributed systems and enable different kinds of self-organization. This chapter is based on my keynote at IJCCI 2019 with the purpose to provide a brief overview of the significance and power of swarm intelligence in coping with some typical distributed systems' problems as well as findings about how and in which use cases the principles of self-organization can contribute to reduce software complexity.

Keywords Self-organization · Swarm intelligence · Distributed systems · Complexity

1 Introduction

Distributed systems develop rapidly and become more and more complex. They usually contain huge number of heterogeneous and mobile nodes, so heterogeneity could be identified as one of the main challenges. When integration of multiple systems is needed, the following issues should be taken in consideration: they differ in their capabilities in terms of integration, have disparities in data, use and support different technologies and standards, and they could be on different platforms [16].

V. Šešum-Čavić (✉)
Institute of Information Systems Engineering, Faculty of Informatics, TU Wien, Argentinierstr. 8, 1040 Vienna, Austria
e-mail: vesna@complang.tuwien.ac.at

J. J. Merelo et al. (eds.), *Computational Intelligence*, Studies in Computational Intelligence 922, https://doi.org/10.1007/978-3-030-70594-7_5

Distributed systems are forced to integrate other software systems and components that are often not reliable, exhibit bad performance, and are sometimes unavailable. Such software is typically characterized by a huge problem size concerning number of computers, clients, requests and size of queries, autonomy and heterogeneity of participating organizations, and dynamic changes of the environment.

Therefore, their complexity[1] becomes a critical issue. The system complexity has been widely identified to be an important problem [1, 12]. Ranganathan and Campbell [16] identifies five aspects of distributed system complexity: task-structure complexity, unpredictability, size complexity, chaotic complexity and algorithmic complexity. To cope with huge dynamics and vast number of unpredictable dependencies on participating components, other approaches are demanded. In attacking complexity, [16] proposes self-configuration and self-repair, high-level programming and interaction, and hierarchical organization of systems and concepts. A useful way would be also implementation of autonomously acting components, which are inspired by nature. These components act in a dynamic, ad hoc way and adapt quickly and self-subsistent to both changing requirements and dynamically evolving system states caused through the interplay and contribution of the many components towards a global goal.

The unavoidable complexity cannot be eliminated, but it can be shifted. Among some well-known tools in coping with the complexity (e.g., abstraction, decoupling, decomposition), a self-organizing approach represents one promising way. Certainly, self-* systems will not be able to adapt to all possible events, but they promise a good perspective to deal with complexity. Herrmann [9] depicts the necessity for self-* mechanisms in distributed systems.

1.1 Self-organization

Researchers have experimented with different paradigms in order to achieve the main properties of self-* systems. Self-* appears in systems without interventions by external directing influences (instructions from a "supervisory leader" or an order imposed on them in many different ways—various directives, recipes, templates) and forms patterns through interactions among their components [3, 10]. Although a functional structure appears and maintains spontaneously, complex systems are not arbitrarily regulated, but ordered in a very organized way. This organization is not built into the system at its origin. It emerges in a sequence of self-organizing processes that include spontaneous transitions into new states of higher organizational complexity. Patterns are well organized structures [3] and can refer to an arrangement of objects both in space (e.g., a zebra's coat) and in time (e.g., firefly flashing). A self-organizing system possesses multiple interdependent components that cooperate in self-initiated interactions [10] through which an information exchange is done.

[1] Note that there are no standard, generally accepted definitions of complexity.

Self-organization in a system appears at different levels (from the lowest level to the highest one), and each of these levels can exhibit their own self-organization. Interacting components are constantly changing their state. "Decisions" and consequently changes are local (e.g. in an ant colony, each ant "decides" by its own which path it will choose). Also, components only interact with their immediate "neighbours". A mutual dependency implies that changes are not arbitrary: some relative states are "preferable", in sense that they will be reinforced or stabilized (like those paths in an ant colony where there are more pheromone), while others are eliminated. The components of the lowest level produce their own emergent properties (patterns) and form the building blocks for the next higher level of organization, with different emergent properties, and this process can further proceed to higher levels in turn.

Most of dynamic systems are metastable possessing many attractors as alternative stable positions. A "noise" (fluctuations) in a system allows the system to escape one basin and to enter another, leading the system to the optimal organization. The basic mechanism underlying self-organization is the variation that governs any dynamic system and allows for exploring of different regions in a state space until it happens to reach an attractor—a preferred position of the system. Thus, increasing variation, i.e., adding "noise" to the system implies that the exploration of a state space will be emphasized, accelerated and deepened. Reaching the attractor, the system comes to the stable state. A further exploration of new state space positions can be continued, if random changes are introduced, which can cause the system to move towards a new attractor [10]. Mathematically speaking, it is possible to have several local optima, but only one global optimum. The self-organization mechanisms have a fully distributed characteristic in a dynamical system, i.e., it must be distributed over all participating components.

We cannot "invent" new forms of self-*: it already exists around us. However, we can learn from biologically-based mechanisms and try to transfer and implement such mechanisms into software systems. Such systems have the following advantages over traditional systems: robustness, flexibility, capability to function autonomously, while demanding a minimum of supervision, and spontaneous development of complex adaptations without need for detailed planning. In mapping, software agents usually play the role of particular swarm individuals (e.g., ants, bees, etc.) and "perform" self-* actions characteristic for the respective swarm colony. All these mechanisms are characterized by a huge number of different environmental parameters influencing the behaviour of artificial swarms.

Although self-* approach is attractive and promising, proven to cope with complexity, the starting question is how to determine whether or not to apply principles of self-organization on a particular use case. First, it is necessary to discern what kind of complexity exists in a particular problem. According to that information, a conclusion can be made about what self-* mechanisms or principle could be suitable for a particular case. For example, if a considered problem possesses programming complexity (and additionally a system itself is rather heterogeneous, like a distributed heterogeneous system), then a high level of autonomy and decoupling is necessary to show some success in coping with this complexity.

1.2 Complexity in Application Scenarios

The sources of complexity in the application scenarios presented in Sect. 2.1 are:

1. amount of resources, i.e., the huge amount of distributed components that must interplay in a global solution,
2. type of resources, i.e., heterogeneity,
3. large number of interactions of the various elements of the software,
4. huge problem size, clients, requests, size of queries etc.,
5. autonomy of organizations,
6. dynamic changes of the environment.

According to these sources, different types of complexity can be discerned: point 1. is a pure computational complexity, points 2. and 3. address programming complexity, point 4. refers to both computational and programming complexity, whereas 5. and 6. are the consequences of features of complex adaptive system.

In the analysis of computational complexity [8], two well-known types appear:

- time complexity—the length of time it takes to find a solution or complete a process as a function of the size of input;
- space complexity—the amount of physical storage required for a system to perform a certain operation, i.e., to solve an instance of the problem as a function of the size of input.

Every task[2] can contain subtasks. The order of complexity of the task is determined through analyzing the demands of each task by breaking it down into its constituent parts [5]. Tasks vary in complexity in two ways: horizontal (involving classical information) or vertical, i.e., hierarchical (involving hierarchical information). Horizontal complexity is the amount of information in simple quantitative terms within a task and consists of the number of different responses that have to be performed [5]. Hierarchical complexity refers to the number of recursions that the coordinating actions must perform on a set of primary elements. The actions at a higher order of hierarchical complexity: (a) are defined in terms of actions at the next lower order of hierarchical complexity; (b) organize and transform the lower-order actions; (c) produce organizations of lower-order actions that are qualitatively new and not arbitrary, and cannot be accomplished by those lower-order actions alone [5].

Example: Consider the action A_1 of evaluating $a + b$ and the action A_2 of evaluating $(a + b) + c$. The horizontal complexity of A_1 is smaller than the horizontal of A_2 since the action of addition is executed less often in A_1 than in A_2. On the other hand, because A_1 differs from A_2 only in how many times addition is executed, but not in the organization of the addition, both actions have the same hierarchical complexity.

So, in the presented application scenarios (Sect. 2.1), the above mentioned types of complexity can be observed: programming and computational, in which both time and space complexity are present; additionally hierarchical complexity is present.

[2]The notion of task is used here as an example, and could be generalized with the notion of process.

1.3 Measurement of Complexity

Researchers from different areas of science like biology, computer science, finance, etc., define different measures of complexity for each respective field. Lloyd [14] present a categorization of complexity measures by defining common questions for all problems:

1. how hard is to describe?
2. how hard is it to create?
3. what is its degree of organization?

A general form of self-organization measurement does not exist. For example, in [4], the mechanism of "brood sorting" is used and spatial entropy is proposed as a measure of self-organization.

In selected use-cases, the measurement of self-organization, i.e., how good the single contributors (bees, ants, …) organize themselves is realized by means of specially constructed functions (e.g., the suitability function in Sect. 3.1). Higher values of these functions denote the better self-organization in the presented systems. Computational complexity is tracked in time.

2 Swarm Intelligence in Distributed Systems

Swarm intelligence possesses distributive and autonomous properties and represents a self-organizing biological system. Every individual in the population makes local decisions, and acts in a decentralized manner. A communication of "knowledge" between individuals is done without any supervisor. Therefore, swarm-inspired algorithms play an important role in the design of self-organizing software for distributed systems. They have a broad spectrum of application areas, support the optimization and robustness of highly dynamic distributed systems, fast adaptation to changes by learning from history and enable different kinds of self-organization. For example, they provide primitives for continued execution when nodes or the network communication fail, when nodes are added or removed during execution, or even in situations when the application should be upgraded "on-the-fly" without interrupting execution.

2.1 Some Selected Distributed Systems' Use-Cases

Some typical distributed systems problems (load balancing, load clustering, information placement and retrieval in heterogeneous networks, distributed routing, peer clustering) have been successfully treated by swarm intelligence.

(1) **Load Balancing** can be described as finding the best possible workload (re)distribution and addresses ways to transfer excessive load from busy

(overloaded) nodes to idle (under-loaded) nodes (Fig. 1). Load Balancing can take place at local node level allocating load to several core processors of one computer, as well as at network level distributing the load among different nodes. Šešum-Čavić and Kühn [18] explains for the first time how bee intelligence can be mapped to dynamic load balancing.

(2) **Load Clustering** deals with clustering of work loads in a computer system. It tries to make further optimizations of the load distribution based on the content of the load items (Fig. 2). A single load item can be described as a task that consists of several attributes (e.g. a certain priority), has a payload, a dynamic life cycle and is handled by a computer or processor. Among different clustering and classifying algorithms (K-Means, Fuzzy C-Means, Genetic K-Means, Hierarchical Clustering, K-Nearest Neighbor, Decision Trees), a usage of ant intelligence in dynamic load clustering is demonstrated [13].

(3) **Information Placement and Retrieval in Heterogeneous Networks**. Šešum-Čavić and Kühn [17] deals with data placement and retrieval in the internet (Fig. 3). Unstructured peer-to-peer overlay network technologies are combined with swarm intelligence (ant intelligence, bee intelligence and slime molds). It is proven that a good query capability with good scalability can be achieved by using swarm-based algorithms.

(4) **P2P Streaming**. Further, the previous use case 3 is extended to streaming in fully decentralized P2P networks [19]. It addresses need to create a P2P application which combines video on-demand streaming and user collaboration

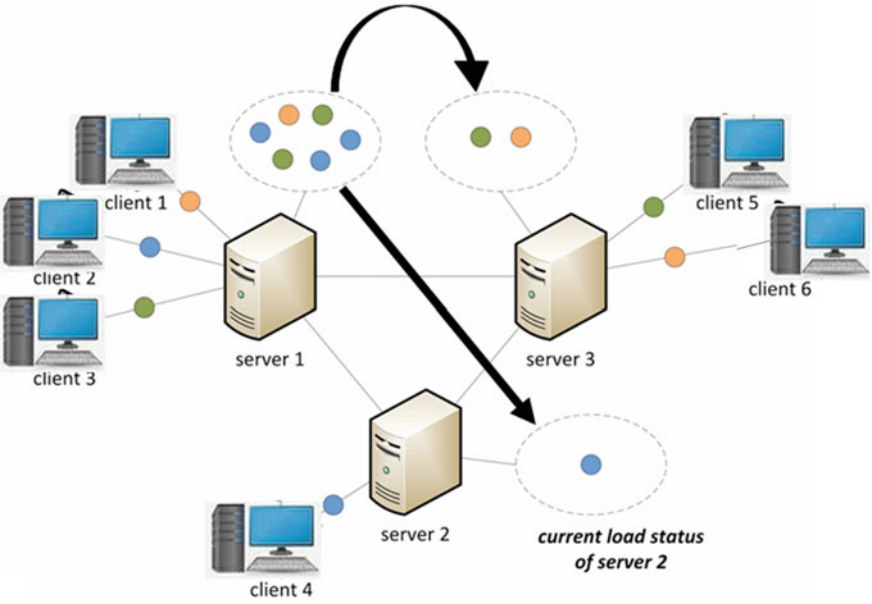

Fig. 1 Dynamic load balancing

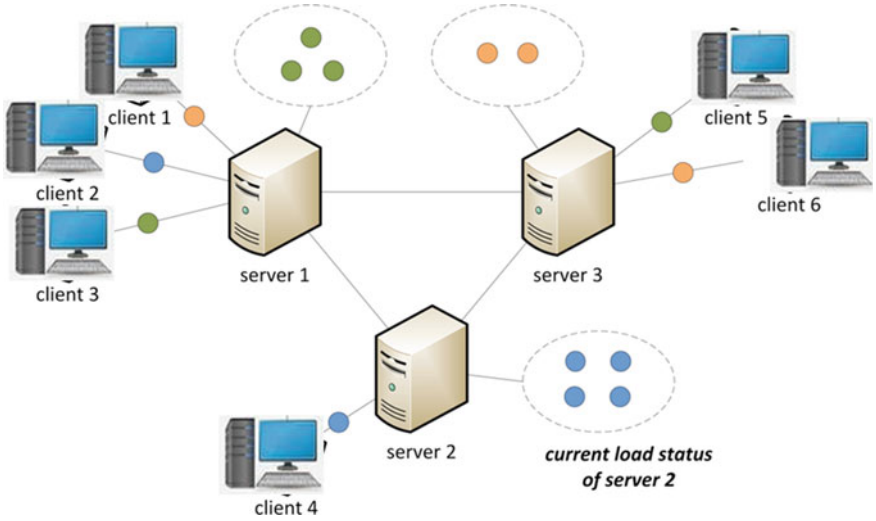

Fig. 2 Dynamic load clustering

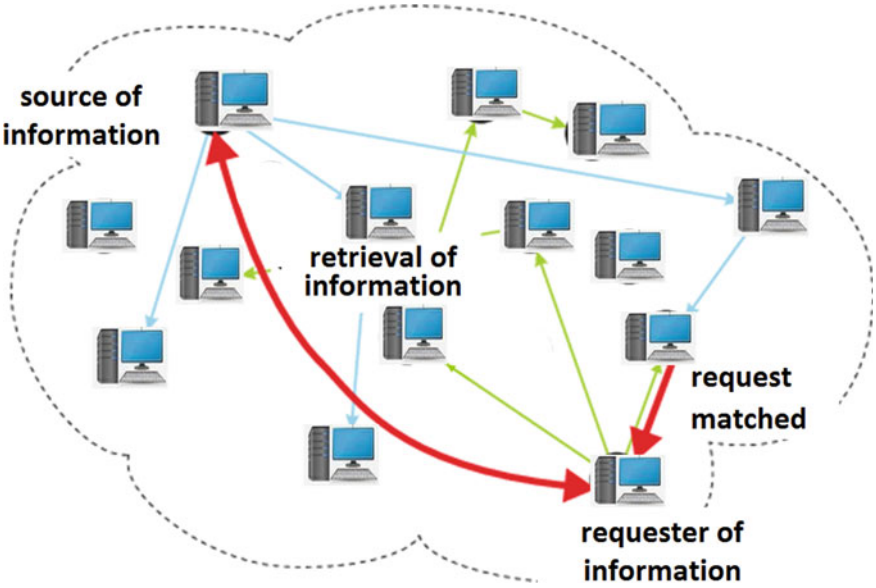

Fig. 3 Information retrieval

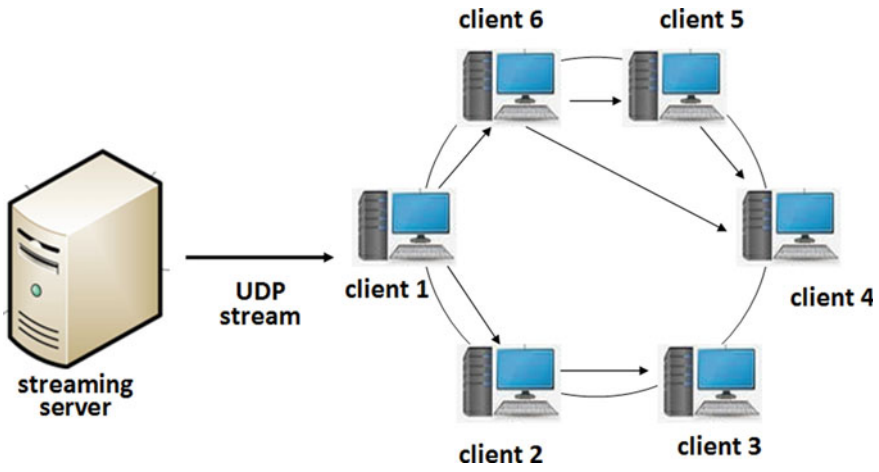

Fig. 4 P2P streaming

(Fig. 4). P2P applications that support the streaming delivery method rely on hybrid approaches, and therefore, are not fully decentralized. Besides ant intelligence and bee intelligence, the lookup mechanism used includes usage of a slime mold intelligence that is adapted for this use case as well as bark beetle intelligence [21] that represents designing a new simple, effective swarm-based algorithm.

(5) **Distributed Routing**. Šešum-Čavić et al. [20] presents modelling of the life-cycle of cellular slime moulds and bee-behaviour based on the foraging mechanism of honey bees in order to create fully distributed routing algorithms for unstructured P2P networks (Fig. 5). A modelling and adaptation of slime mould intelligence is done for the first time for routing in unstructured P2P networks. Bee intelligence was already applied to the routing problem in general [22]. However, in [20], another type of mapping and adaptation is proposed.

2.2 Algorithm Recommendation for Selected Use Cases

The selected problems numbered in Sect. 2.1 were treated by using different approaches and types of algorithms (conventional and swarm intelligent). The details of specific adaptations and implementations can be found in [13, 17–21]. The obtained results proved the significance of usage swarm intelligence approach in complex, dynamical distributed systems' problems. In this subsection, a kind of recommendation algorithms for selected use cases is presented as a sum-up of obtained results (Table 1).

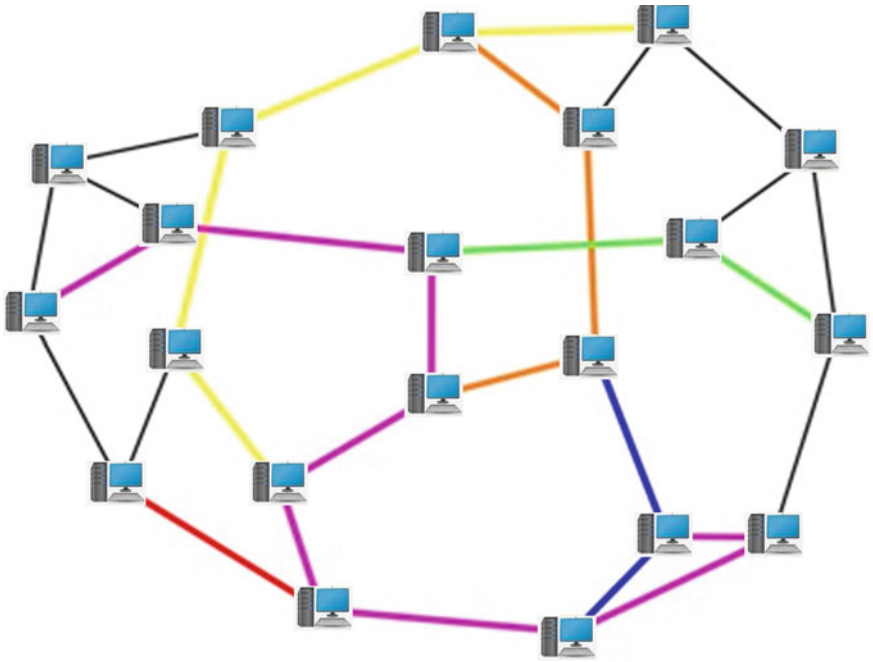

Fig. 5 Distributed routing

3 An Illustration: Bee Algorithm for Dynamic Load Balancing

For sake of illustration, a mapping and adaptation of bee algorithm for dynamic load balancing [18] is shortly reviewed. As this scenario refers to unstructured P2P networks, a formalization of P2P network model is presented. Further, some theoretical concepts of the presented bee algorithm are discussed.[3]

3.1 Bee Algorithm

In a honeybee colony, bees have different roles: foragers, followers, and receivers. A functioning of a bee colony relies on two main strategies: (*) navigation—searching for nectar in an unknown landscape; a forager searches for a flower with good nectar and after finding and collecting, it returns to the hive and unloads the nectar, and (*) recruitment—a forager performs the so-called "waggle dance", i.e., it communicates the knowledge about the visited flowers (quality, distance and direction) to other bees (Fig. 6). A follower randomly chooses to follow one of the foragers and visits the

[3]More details incl. benchmarking results can be found in [18].

Table 1 A sum-up for algorithm recommendation in selected use-cases

Scenario	Recommended algorithm(s) or combination of algorithms	Metric
Load balancing	– Both combinations BeeAlgorithm/Sender and MinMaxAS/MinMaxAS were equal good in the chain topology – Both combinations BeeAlgorithm/Sender and MinMaxAS/RoundRobin were equal good in the ring topology – Both combinations BeeAlgorithm/BeeAlgorithm and GA/AntNet were equal good in the star topology – A combination RoundRobin/BeeAlgorithm was the best in the full topology – Bee algorithms play a significant role in almost each topology, as the best obtained results in each topology are based on bee algorithms either used inside subnets or used between subnets or both	Absolute execution time
Load clustering	– From the group of clustering algorithms, Hierarchical Clustering obtained the best results, whereas from the group of classification algorithms the Ant-Miner algorithm was the best – The combination of the Hierarchical algorithm with any other, except the Genetic K-Means algorithm, leads to a good execution time. The best result was delivered by the combination of the Hierarchical and Fuzzy C-Means algorithm. The Hierarchical Clustering showed the best results in a small network with only one client that supplies load. For large and more complex networks, an intelligent approach with an appropriate similarity function will help	Absolute execution time

(continued)

Table 1 (continued)

Scenario	Recommended algorithm(s) or combination of algorithms	Metric
Information placement and retrieval	– Random/AntNet algorithm is better than Random/MinMaxAS; the possible reason for that could be the fact that Random/AntNet better supports dynamic processes – Bee algorithm obtained the best results especially on large instances	Absolute execution time
P2P streaming	– Bark Beetles algorithm, Physarum Polycephalum algorithm, Gnutella flooding, k-Walker, AntNet and Dd-slime mold algorithms are compared – Absolute time: for small replication rate (2%), all network sizes, Physarum Polycephalum algorithm outperforms the other algorithms; for bigger replication rate (16%), all network sizes, Bark Beetles algorithm outperforms the other algorithms – Average message per node: for all replication rates, all network sizes, Bark Beetles algorithm outperforms the other algorithms – Success rate: for small replication rate (2%), network sizes of 50 and 100 nodes, Bark Beetles algorithm has comparable success rate as Gnutella; for bigger replication rate (16%), all network sizes, Bark Beetles algorithm has 100% success rate	Absolute execution time, success rate, and average messages per node

(continued)

Table 1 (continued)

Scenario	Recommended algorithm(s) or combination of algorithms	Metric
Routing in P2P	– Slime Mold routing algorithm (SMNet) outperformed all other benchmarked routing algorithms (AntNet, BeeHive, Gnutella, k-Random Walker) regarding the average delivery delay of data packets with growing amount of network nodes and data packet traffic – Bee routing algorithm (BeeNet) took the overall second place right after SMNet	Data packet delivery ratio, average data packet delay, average data packet hop count, and routing overhead messages

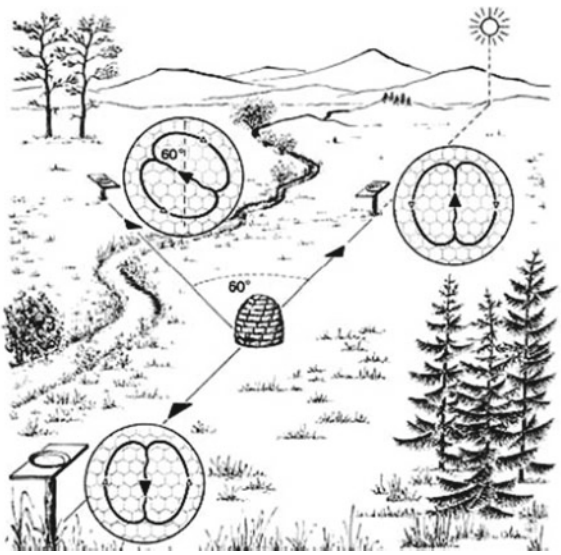

Fig. 6 A honeybee colony in nature [2]

"advertised" flower. Foragers and followers can change their roles in the next step of navigation. A receiver processes the nectar in the hive.

A software agent plays the role of bee and resides at a particular node. A node consists of exactly one hive and one flower in its environment. A task is one nectar unit in a flower. Following situations are possible: (*) there are more nectar units in a flower, (*) a flower is empty (in that case, it is not removed from the system). A new task can be put at any node in the network. A hive has k stationary bees (receivers) and l outgoing bees (foragers and followers). Initially, all outgoing bees are foragers.

Foragers scout for a "partner" node of the node that they belong to, i.e., a particular resource for load balancing to get or put work load from/to it. Further, they inform and recruit followers. Thus, the main actors are foragers and followers as receiver bees process tasks at their node and have no influence on the algorithm itself [18].

The goal is to find the best partner node (determined by means of a suitability function) by taking the best path (here defined as the shortest path). A navigation strategy determines which node will be visited next and is realized by a state transitions rule [23]:

$$P_{ij}(t) = \frac{[\rho_{ij}(t)]^{\alpha} \cdot [1/d_{ij}]^{\beta}}{\sum_{j \in A_i(t)} [\rho_{ij}(t)]^{\alpha} \cdot [1/d_{ij}]^{\beta}} \tag{1}$$

where d_{ij} is the heuristic distance between i and j, α is a binary variable that turns on/off the arc fitness influence, and β is the parameter that controls the significance of a heuristic distance, and $\rho_{ij}(t)$ is the arc fitness from node i to node j at time t and is calculated in the following way: $\rho_{ij} = 1/k$, where k is the number of neighbouring nodes of node i in case of forager, whereas in case of follower [23]

$$\rho_{ij}(t) = \begin{cases} \lambda & if\ j \in F_i(t) \\ \frac{1-\lambda \cdot |A_i(t) \cap F_i(t)|}{|A_i(t)| - |A_i(t) \cap F_i(t)|} & if\ j \notin F_i(t) \end{cases} \quad \forall j \in A_i(t),\ 0 \leq \lambda \leq 1 \tag{2}$$

where $A_i(t)$ is the set of allowed next nodes, i.e., the set of neighbouring nodes of node i, and $F_i\ (t)$ is the set of favoured next nodes recommended by the preferred path.

During the recruitment, bees communicate using the following parameters: path (distance), and quality of the solution. Therefore, fitness function f_i for a particular bee i can be derived as [18]:

$$f_i = \frac{1}{H_i}\delta \tag{3}$$

where H_i is the number of hops on the tour, and δ is the suitability function. The colony's fitness function f_{colony} is the average of all fitness functions (for n bees):

$$f_{colony} = \frac{1}{n}\sum\nolimits_{i=1}^{n} f_i \tag{4}$$

If bee i finds a highly suitable partner node, then its fitness function, f_i obtains a good value. After a trip, an outgoing bee determines how "good it was" by comparing its result f_i with f_{colony}, and based on that decides its next role [15].

Therefore, two following situations can occur [18]:

- if a bee of an under-loaded node searches for a suitable task belonging to some overloaded node, then this bee carries the information about how complex a task the node can accept;
- if a bee of an overloaded node searches for an under-loaded node that can accept one or more tasks from this overloaded node, then it carries the information about the complexity of tasks this overloaded node offers and compares it with the available resource of the current under-loaded node that it just visits.

In both cases, the complexity of the task should be compared with the available resources at a node [18]. For this purpose, the following notions are introduced: task complexity c, host load hl and host speed hs, whereas hs is relative in a heterogeneous environment, hl represents the fraction of the machine that is not available to the application, and c is the time necessary for a machine with $hs = 1$ to complete a task when $hl = 0$. We calculate the argument $x = (c/hs)/(1 - hl)$ of suitability function δ and define it as $\delta = \delta(x)$. For example, when an under-loaded node with high resource capacities is taking work from an overloaded node node, a partner node offering tasks with small complexity is not a good partner as other nodes could perform these small tasks as well. Taking them would mean wasting available resources.

3.2 P2P Network Model

A formalized description of a P2P overlay network 20 is closely related to the definition of a unique identifier of each P2P node. Since P2P overlay networks operate above the physical layer, this unique identifier must not be the physical host address.

Let a P2P overlay network be represented by a graph $G_{P2P} = (V_{P2P}, E_{P2P})$, where the nodes $v \in V_{P2P}$ of the graph represent nodes in a P2P network and the links $e \in E_{P2P}$ represent connections between these nodes. Nodes $v_1, v_2 \in V_{P2P}$ are neighbours, if and only if $\exists\ (v_1, v_2) \in E_{P2P}$. Each node $v_i \in V_{P2P}$, $i = 1,\ldots, n$, $n = | V_{P2P} |$ has a physical address y_{es} and a logical unique identifier x_i, which is known to all nodes, but only neighbours are able to map the logical identifier x to the physical host address y and therefore, exchange packets directly:

$$m(c, x_e) = \begin{cases} y_e & if\ e \in neighbours(c) \\ x_e & otherwise \end{cases}$$

where $c, e \in V_{P2P}$.

Intelligent swarm agents (e.g., bees) may know the physical address y_s of their source node in addition to the logical identifier x_s, and therefore, may return directly to their source.

3.3 Convergence

References [6, 11] investigate the convergence of Bee Colony Optimization algorithm and prove that the current best solution converges to one of the optimal solutions (with the probability one) as the number of iterations increases.

We provide a convergence in value of the bee algorithm from Sect. 3.1. For this purpose, pre-assumptions and formalization are taken from [7]:

1. $G_{P2P}=(V_{P2P}, E_{P2P})$ is a graph of n nodes and links between these nodes (nodes are not necessarily fully connected in the load balancing scenario);
2. $S, f, \Phi)$, where S is the set of candidate solutions, f is the objective function, Φ is the set of constrains that defines the set of feasible solutions; the goal is to find an optimal solution s_{opt}; Θ is the finite set of states of the problem, $\theta =< v_i, v_j, \ldots, v_h, \ldots >$, $|\theta|$ is the number of nodes in a sequence, $|\theta| \leq n$; Θ^* is the set of feasible states, $\Theta^* \subseteq \Theta$;
3. for the time being, static scenarios in this theoretical explanation are considered.

The probability rule of navigation strategy in construct solution phase could be described as:

$$P(c_{h+1} = j|x_h) = \frac{F_{ij}(\rho)}{\sum_{j \in A_i} F_{ij}(\rho)} \tag{5}$$

where F_{ij} is some non-decreasing function, $F_{ij}(z) = z^{\alpha}\eta_{ij}^{\beta}$.

A recruitment phase represents the exchange of knowledge about the path length (distance, which is expressed as the number of hops) and quality of the solution (measured by some similarity function, δ). This phase is described by Eq. 3.

In the following, a new result is derived as the consequence of the similar result that considers convergence of one group of Ant System Algorithms in which, for example, Min-Max Ant System belongs to 7. Therefore, the next corollary is inspired and based on one theorem from 7 that proves convergence in value of Min-Max Ant System. The theorem says that when using a fixed positive lower bound on the pheromone trails finding the optimal solution is guaranteed for this specific group of algorithms. The next proof is based on some specifics of the bee algorithm and some general issues that could be found in the proof of convergence in value of Min-Max Ant System as well 7.

Corollary: If $P(k)$ is the probability that the bee algorithm finds an optimal solution at least once within the first k iterations, then $\lim_{k \to +\infty} P(k) = 1$.

Proof From Eq. 2, it follows that the arc fitness ρ for a follower bee belongs $\rho \in \{\frac{1-\lambda}{l-1}, \lambda\}$, where λ is the probability of choosing the preferred path and l is the number of neighbouring nodes of a particular node. If the case for a forager bee is added, that means $\rho \in \{\frac{1-\lambda}{l-1}, \lambda, \frac{1}{l}\}$. So, for the given network values of arc fitness can have a finite number of values and it values stay in some closed interval $[\rho_{min}, \rho_{max}]$. The lower bound is positive and fixed for the given network. Therefore, any feasible choice

from Eq. 1 for any partial solution x_h is made with the probability:

$$p_{\min} \geq \frac{\rho_{\min}^{\alpha}}{(n-1)\rho_{\max}^{\alpha} + \rho_{\min}^{\alpha}} \tag{6}$$

Any solution (incl. the optimum solution) can be generated with the probability:

$$p > \left(\frac{\rho_{\min}^{\alpha}}{(n-1)\rho_{\max}^{\alpha} + \rho_{\min}^{\alpha}} \right)^{m} > 0 \tag{7}$$

where m is the maximum length of a sequence. From this fact, it follows that $P(k) = 1 - (1-p)^k$. For every arbitrarily small $\varepsilon > 0$, $P(k) \geq 1 - \varepsilon$. That means: $\lim_{k \to +\infty} P(k) = 1$. ■

The corollary explains the following. In the bee algorithm, the values that are assigned to arcs are the values of arc fitness, ρ_{ij}. Some of these values will be implicitly reinforced by learning of the other hive mates via waggle dance (i.e., a recruitment process). The fact how "strong" is the recruitment of a particular bee depends on the values of suitability function and the path length. The higher the value of suitability function and the lower the path length, the stronger the recruitment is. The bee algorithm forces the best-so-far solution, and uses implicit maximum value of ρ_{max} (which is directly implied by f_{max} from the best-so-far solution). Further, the value of λ is initialized to the upper limit (λ_0), so the minimum value ρ_{min} will be reached in:

(a) $\frac{1-\lambda_0}{l-1}$, for any case
(b) $\frac{1-\lambda_0}{n-1}$, for the case with fully connected nodes.

Also, any feasible solution can be constructed with a nonzero probability. If we assume that connection (i, j) does not have the largest probability to be chosen (i.e., j does not belong to set F_i), then the probability of choosing this connection is $\frac{1-\lambda}{l-1}$.

4 Conclusion

An extreme and raising complexity characterizes nowadays distributed systems. Self-* approaches represent a promising way to cope with complexity. However, self-organization in different forms already exists around us, from nature to social and economic organizations. Especially inspiring self-organization forms are those ones that could be found in nature, e.g., different types of swarm intelligence. Thus, swarm-intelligent algorithms significantly contribute to the design of self-organizing software for distributed systems. In this chapter, some typical distributed use cases, which are successfully treated by swarm intelligent approaches, are shortly overviewed (incl. the list of the most successful swarm-based algorithms applied in these use cases). As an illustration, one use case is selected—dynamic load balancing, and

an application of bee intelligence onto this problem is reviewed. In order to explain better the behaviour of bee algorithm, its theoretical aspects are discussed.

Future work includes:

- standardizing methodology for a fair evaluation of algorithms in distributed systems' use cases;
- a theoretical evaluation of swarm-intelligent algorithms, an explanation of "why specific methods work well on specific problems" and the analysis of algorithm's behavior. Although this is a challenging task, for certain metaheuristics, theoretical work regarding convergence is partially established with some encouraging results, whereas for many others, no theoretical background exists.

References

1. Asprey, W., et al.: Conquer system complexity: build systems with billions of parts. In: CRA Conference on Grand Research Challenges in Computer Science and Engineering, pp. 29–33 (2002)
2. Barth, F.: Insects and Flowers: The Biology of a Partnership. Princeton University Press, Princeton (1982)
3. Camazine, S., Deneubourg, J., Franks, N.R., Sneyd, J., Theraulaz, G., Bonabeau, E.: Self-Organization in Biological Systems. Princeton University Press, Princeton (2003)
4. Casadei, M., Menezes, R., Viroli, M., Tolksdorf, R.; Self-organized over-clustering avoidance in tuple-space systems. In: IEEE Congress on Evolutionary Computation (2007)
5. Commons, M.L., Goodheart, E.A., Dawson, T.L.: Psychophysics of stage: task complexity and statistical models. In: International Objective Measurement Workshop at the Annual Conference of the American Educational Research Association (1997)
6. Davidovic, T., Teodorovic, D., Selmic, M.: Bee colony optimization part I: the algorithm overview. YUJOR **25**(1), 33–56 (2015)
7. Dorigo, M., Stützle, T.: Ant Colony Optimization. MIT Press (2004)
8. Fortnow, L., Homer, S.: A short history of computational complexity. Bull. EATCS **80**, 95–133 (2003)
9. Herrmann, K.: MESH Mdl — a middleware for self-organization in ad hoc networks. In: 23rd International Conference on Distributed Computing Systems (2003)
10. Heylighen, F.: The science of self-organization and adaptivity. In: Kiel, L.D. (ed.) Knowledge Management, Organizational Intelligence and Learning, and Complexity. The Encyclopedia of Life Support Systems. EOLSS Publishers, Oxford (2001)
11. Jakšić-Krüger, T., Davidović, T., Teodorović, D., et al.: The bee colony optimization algorithm and its convergence. Int. J. Bio-Inspired Comput. **8**(5), 340–354 (2016)
12. Kephart, J., Chess, D.: The vision of autonomic computing. IEEE Comput. **36**(1), 41–50 (2003)
13. Kühn, E., Marek, A., Scheller, T., Šešum-Čavić, V., Vögler, M.: A space-based generic pattern for self-initiative load clustering agents. In: 14th International Conference on Coordination Models and Languages (2012)
14. Lloyd S.: Measures of complexity: a nonexhaustive list. IEEE Control Syst. (2001)
15. Nakrani, S., Tovey, C.: On honey bees and dynamic server allocation in the internet hosting centers. Adapt. Behav. **12**, 223–240 (2004)
16. Ranganathan, A., Campbell, R.H.: What is the complexity of a distributed computing system? Complexity **12**(6), 37–45 (2007)
17. Šešum-Čavić, V., Kühn, E.: A swarm intelligence appliance to the construction of an intelligent peer-to-peer overlay network. In: 4th International Conference on Complex, Intelligent and Software Intensive Systems (2010)

18. Šešum-Čavić, V., Kühn, E.: Self-organized load balancing through swarm intelligence. In: Next Generation Data Technologies for Collective Computational Intelligence. Studies in Computational Intelligence, vol. 352, pp. 195–224. Springer (2011)
19. Šešum-Čavić, V., Kühn, E., Kanev D.: Bio-inspired search algorithms for unstructured P2P overlay networks. Swarm Evolut. Comput. **29**, 73–93 (2016). Elsevier
20. Šešum-Čavić, V., Kühn, E., Zischka, S.: Swarm-inspired routing algorithms for unstructured P2P networks. In: Int. J. Swarm Intell. Res. IJSIR **9**(3) (2018). Article 2
21. Šešum-Čavić V., Kühn E., Fleischhacker L.: Efficient search and lookup in unstructured P2P overlay networks inspired by swarm intelligence. IEEE Trans. Emerg. Top. Comput. Intell. (in press)
22. Wedde, H.F., Farooq, M., Zhang, Y.: BeeHive: an efficient fault-tolerant routing algorithm inspired by honey bee behaviour. Ant Colony Optim. Swarm Intell. 83–94 (2004)
23. Wong, L.P., Low, M.Y., Chong, C.S.: A bee colony optimization for travelling salesman problem. In: 2nd Asia International Conference on Modelling & Simulation, pp. 818–823 (2008)