Chapter 8
Self-Organized Load Balancing through Swarm Intelligence

Vesna Šešum-Čavić and Eva Kühn

Abstract. The load balancing problem is ubiquitous in information technologies. New technologies develop rapidly and their complexity becomes a critical issue. One proven way to deal with increased complexity is to employ a self-organizing approach. There are many different approaches that treat the load balancing problem but most of them are problem specific oriented and it is therefore difficult to compare them. We constructed and implemented a generic architectural pattern, called SILBA, which stands for "self-initiative load balancing agents". It allows for the exchanging of different algorithms (both intelligent and unintelligent ones) through plugging. In addition, different algorithms can be tested in combination at different levels. The goal is to ease the selection of the best algorithm(s) for a certain problem scenario. SILBA is problem and domain independent, and can be composed towards arbitrary network topologies. The underlying technologies encompass a black-board based communication mechanism, autonomous agents and decentralized control. In this chapter, we present the complete SILBA architecture by putting the accent on using SILBA at different levels, e.g., for load balancing between agents on one single node, on nodes in one subnet, and between different subnets. Different types of algorithms are employed at different levels. Although SILBA possesses self-organizing properties by itself, a significant contribution to self-organization is given by the application of swarm based algorithms, especially bee algorithms that are modified, adapted and applied for the first time in solving the load balancing problem. Benchmarks are carried out with different algorithms and in combination with different levels, and prove the feasibility of swarm intelligence approaches, especially of bee intelligence.

1 Introduction

The IT-industry continuously faces a rapid increase in the complexity of software systems. New requirements, a large number of interacting components with

Vesna Šešum-Čavić · Eva Kühn
Technical University Vienna, Institute of Computer Languages,
Argentinierstrasse 8, 1040 Wien, Austria
{vesna,eva}@complang.tuwien.ac.at

internal states defined by many thousands of parameters, applications that rely on other unreliable systems, and many components tied together are only a few reasons that impose the necessity of finding new approaches for software systems. Main factors that determine software complexity are:

- Huge amounts of distributed components that interplay in a global solution,
- Problem size like number of computers, clients, requests, size of queries etc.,
- Heterogeneity,
- Autonomy of organizations, and
- Dynamic changes in the environment.

Distributed software systems are forced to integrate other software systems and components that are often not reliable, exhibit bad performance, and are sometimes unavailable. These challenges are so fundamental that, the usually taken approach to control distributed components across enterprise boundaries through one central and predefined coordinator software, reaches its technical and conceptual limits. The huge number of unpredictable dependencies on participating components cannot be coped with any more in the traditional way, namely through one central coordinator that implements the entire business logic and that possesses the complete picture of the distributed environment and all possible exceptions. A very useful concept in the adaptation of complex systems is *self-organization*. Certainly, self-organizing systems will not be able to adapt to all possible events, but they have proven to pose a good perspective to deal with complexity through self-organization, self-repairing, self-configuring, self-grouping, self-learning, self-adaptation, etc.

In this chapter, we consider the problem of load balancing (LB) in the light of the above mentioned challenges of today's systems. LB 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. Dynamic LB should improve the performance of the overall distributed system and achieve the highest level of productivity.

1.1 Related Approaches

There are many different approaches that cope with LB. The first group consists of different conventional approaches without using any kind of intelligence, e.g.: Sender Initiated Negotiation and Receiver Initiated Negotiation [33], Gradient Model [22], Random Algorithm [38], and Diffusion Algorithm [7]. In Sender algorithm, LB is initiated by the over-loaded node. This algorithm has a good performance for low to moderate load levels while in Receiver algorithm, LB is initiated by the under-loaded node and this algorithm has a good performance for moderate to heavy load levels. Also the combination of these two algorithms

(Symmetric) is possible. The Gradient Model is based on dynamically initiated LB requests by the under-loaded node. The result of these requests is a system wide gradient surface. Overloaded nodes respond to requests by migrating unevaluated tasks down the gradient surface towards under-loaded nodes. In Random Algorithm each node checks the local workload during a fixed time period. When a node becomes over-loaded after a time period, it sends the newly arrived task to a randomly chosen node without taking in consideration whether the target node is over-loaded or not. Only the local information is used to make the decision. The principle of diffusion algorithms is keeping the process iterate until the load difference between any two processors is smaller than a specified value. The *second* group includes theoretical improvements of LB algorithms using different mathematical tools and estimations [2] without focusing on implementation and benchmarks. The *third* group contains approaches that use intelligent algorithms like evolutionary approaches [5], and ant colony optimization approaches [12]. Evolutionary approaches use the adjustment of some parameters specific for evolutionary algorithms to achieve the goal of LB. Ant colony optimization is used in [12] for a graph theoretic problem formulated from the task of computing load balanced clusters in ad hoc network. The intelligent algorithms from the last group showed promising results. However, they still need improvement concerning experience in the tuning of algorithms, the quality of solution they provide, scalability, the provisioning of a general model, and flexibility. In [21], non-pheromone-based (bee intelligence) versus pheromone-based algorithms are compared. Their conclusion is that the former are significantly more efficient in finding and collecting food.

These approaches mainly try to improve only one of the components of the whole LB infrastructure, namely the LB algorithm itself. A comprehensive classification of different LB approaches is given in [19], where we refer to the problem as the lack of a general framework, autonomy, self-* properties, and arbitrary configurations, and introduced a LB pattern, i.e. a software building block that abstracts the LB problem and that can be re-used in many different situations by simply configuring it termed SILBA (self-initiative load balancing agents) that addresses the following issues:

General Framework: Existing LB approaches are very problem specific. As there is no "one-fits-all" solution, in order to find the best solution for a problem, a general framework is needed that allows for testing and tuning of different LB algorithms for a given problem and environment. The SILBA architectural pattern is agile [24] and supports an easy and dynamic exchange of pluggable algorithms as well as combinations of different algorithms with the goal to ease the selection of the best algorithm for a certain problem scenario under certain conditions. Note that a framework itself doesn't solve the LB problem but serves as necessary basement for testing LB algorithms.

Autonomy and Self- Properties:* Increased complexity of software systems, diversity of requirements, and dynamically changing configurations imply a necessity to find new solutions that are e.g. based on self-organization, autonomic computing and autonomous (mobile) agents. Intelligent algorithms require autonomous agents which are advantageous in situations that are characterized by high dynamics, not-foreseeable events, and heterogeneity.

Arbitrary Configurations: LB can be required to manage the load among local core processors on one node, as well as in a network (intranet, internet, cloud). A general LB framework must be able to cope with all these demands at the same time and offer means to abstract hardware and network heterogeneities.

Our research focuses on a new conception of a self-organizing coordination infrastructure that suggests a combination of coordination spaces, self-organization, adaptive algorithms, and multi-agent technologies[1]. Each of these technologies has some form of self-organization in its incentive. In this chapter, after explaining the SILBA pattern in its basic form that supports LB between nodes in one subnet and briefly describing the obtained results, as a further step, SILBA is extended:

1. to support load balancing on several levels, i.e. not only between agents of the same node, but also between agents of different nodes and possible in different subnets and
2. to allow for combinations of different algorithms on different levels (e.g., swarm intelligence on each level, or swarm intelligence combined with unintelligent algorithms).

Our contributions are summarized in the following points where (1.) concerns our previous work on basic SILBA, and (2.)-(5.) concern extended SILBA:

1. Implementation of different algorithms, fine tuning of parameters and comparison of unintelligent versus intelligent algorithms, by plugging them into the SILBA pattern and benchmarking them: For the intelligent algorithms, we: a) adapted and implemented two ant algorithms, b) adapted and for the first time implemented the concepts of bee intelligence to the LB problem. The novelty includes the mapping and implementation of bee intelligence for the LB problem to improve the quality of the solution and scalability.
2. Realization of LB by extending it to several levels.
3. Construction of different combinations of algorithms for LB.
4. Investigation which combination of algorithms fits best for a particular network topology, and which topologies profit the most from the application of swarm intelligence.
5. Achievement of self-organization through different methods (like swarm intelligence, autonomous agents) in combination.

[1] It is assumed that a reader is familiar with the basic concepts of multi-agent technologies (see, for example [34], for an overview).

2 SILBA Framework

The SILBA framework is based on multi-agents technology and space-based computing. Space-based computing (SBC) is a powerful concept for the coordination of autonomous processes, an easy to use solution that handles the complexity of the interplay of autonomous components in a heterogeneous environment through a high abstraction of the underlying hardware and operating system software [20]. The processes communicate and coordinate themselves by simply reading and writing distributed data structures in a shared space. Although SBC is mainly a data-driven coordination model, it can be adapted and used according to control-driven coordination models. A space offers many advantages: a high level abstraction for developers that allows for hiding complexity, reliable communication, transactions, asynchrony, near-time event notification, scalability and availability [20].

SILBA uses a space-based architecture, called XVSM (extensible virtual shared memory) [16]. It generalizes Linda tuple based communication [11] as well as several extensions to it like reactions [30], programmable behaviour [3], and further coordination laws like priority and user defined match makers [29]. Comparable to Linda, a container represents a shared data space that can be accessed by the operations *read*, *take*, and *write*. Beyond that it can be addressed via an URL, can reference other containers, and is extensible through aspects [15], i.e., code fragments that react to certain events and serve to build higher level behaviour and interfaces on top of a container thus forming more complex coordination data structures. This asynchronous and blackboard based communication model is advantageous to for collaboration of autonomous (multi)agents as it avoids coupling through direct interactions [13] between the agents, especially when mobile agents are assumed [3], [29].

2.1 Basic SILBA

Basic SILBA supports the exchange of algorithms (both unintelligent and intelligent ones) as a test bed to ease the evaluation of the best algorithm for a certain problem scenario under certain conditions. It is based on decentralized control. Self-organization is achieved by using a blackboard based style of collaboration to build up a shared view on the current state. The SILBA pattern is domain independent and can be used at different levels:

- *Local* node level: allocating load to several core processors of one computer – the determining factor for load distribution is the balanced utilization of all cores.
- *Network* level: distributing load among different nodes. This includes load balancing within and between different subnets. One must take into consideration

the time needed for transferring data from a busy node to an idle node and estimate the priority of transferring, especially when the transfer itself requires more time to complete than the load assignment.

The basic components of SILBA are clients, autonomous agents, tasks and policies. Clients request tasks to be executed, i.e. load originates from clients. Different types of autonomous agents operate in a peer-to-peer manner and decide on their own when to pick up or push back work, assuming that the amount of work is changing dynamically. A task can be described as a tuple of the form "(priority, job, description, properties, timeout, answer space)", denoting the priority of a task in absolute terms, the job in a standard format like XML or WSDL, an optional (semantic) description, properties (e.g., whether task's execution mode is "at-most-once" or "best-effort"), a timeout, and a URL of an Internet addressable resource where to write the result of the execution back, that we term answer space to make the protocol stateless and to support not always connected networks. SILBA puts emphasis on two main policies termed transfer policy and location policy. The transfer policy determines whether and in which form a resource participates in load distribution and in that sense determines the classification of resources [33] into the following categories: under-loaded (UL), ok-loaded (OK) and overloaded (OL). The transfer policy is executed by a worker agent autonomously. A worker agent may reject a task it has started and re-schedule it. The location policy determines a suitable partner of a particular resource for LB [33]. The SILBA pattern is composed of three sub-patterns [19]:

The **local node pattern** is responsible for the execution of requests by local worker agents actively competing for work. The basic components of this pattern are: clients, worker agents, load space, and answer space. Load space is a place where new requests are put by clients and information about all worker agents' registrations and the current load status (UL, OK, OL) of a node are maintained. Requests are accessible in either the order they arrived, or by means of other criteria like their priority, the required worker role, or their expiration date. Answer spaces are places where the answers computed by worker agents are put directly (not routed) and where they can be picked up by the corresponding clients.

The **allocation pattern** redirects load between load spaces of different local nodes. The basic components of the allocation pattern are: load space, allocation agents, policies, and allocation space. There are three kinds of allocation agents: arbiter agents, IN agents, and OUT agents. Arbiter agents query the load of the load space and decide about re-distribution of work. They publish this information to the routing space in form of routing requests. Both IN and OUT agents read routing information from the allocation space and pull respectively push work from/to another node in a network to which the current node has a connection. The

IN and OUT allocation agents assume that the information about the (best) partner to/from which to distribute load can be queried from the allocation space. The allocation space holds information about partner nodes as computed by the location policy. This information is queried by the allocation agents and can be either statically configured or dynamically computed by routing agents.

The **routing pattern** executes the location policy according to a particular LB algorithm. The basic components of the routing pattern are: allocation space, routing agents, and routing space. Routing agents perform the location policy by implementing a certain LB algorithm and by communicating with other routing agents of the same type forming a dynamically structured overlay network. The collaboration between routing agents of different nodes is carried out via the corresponding routing spaces of this type. Each kind of routing agents has its own routing space where specific information, required by the applied algorithm, is stored and retrieved (e.g. pheromones for ants, or duration of waggle dance for bees). Eventually, the information about the best or suitable partner nodes is stored in the allocation space where a corresponding IN or OUT allocation agent grabs this information and distributes the load between the local node and its partner node.

The above described patterns can be composed towards more complex patterns by "hooking" them via shared spaces. They must agree on the format of entries stored in these spaces, and on the interaction patterns on these. With SILBA patterns, bi-directional control flows are possible and arbitrary logical network configurations can be easily and dynamically be constructed. Example in Fig. 1 shows four subnets A-D that have different relationships to each other. Nested subnets are allowed and two (or more) subnets might overlap, i.e. have in their intersection 0, 1 or more nodes. Therefore, nodes can belong to one or more subnets, e.g., nodes N1 and N2 are part of one subnet each, whereas N3 belongs to two subnets.

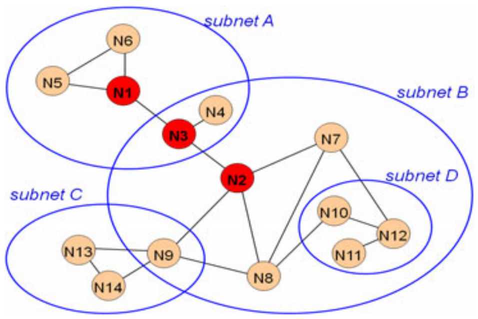

Fig. 1 Topology example.

The XVSM shared data space, which has already been successfully applied in several agents based projects [17], [18] serves as the coordination middleware for SILBA. Shared data structures maintain collaboration information and other LB relevant parameters to tune the algorithms. This indirect communication allows for a high autonomy of agents. Concurrent agents either retrieve, or subscribe to this information being notified in near-time about changes, or modify it. Clients continuously put tasks to any node in the distributed network.

2.2 Extended SILBA

SILBA is designed so that it can be extended towards the remote load balancing as sketched in the following. The main point is that the routing sub-patterns for different levels of load balancing are the same and simply composed towards the desired topology by "connecting" them via shared spaces; all sub-patterns can be parameterized by different algorithms. Each level can apply a different algorithm and load balancing in the entire network occurs through the combination of all algorithms. Fig. 2 represents the realization of nodes N1, N2 and N3 from Fig. 2. Boxes represent processes (clients or agents), and circles represent shared spaces. For simplicity, the three roles of arbiter, IN and OUT agent are represented by one box in the allocation pattern. Sub-patterns are edged with dotted lines. A composition occurs where sub-patterns overlap, in that they jointly access a shared space. SILBA can be composed to support an arbitrary amount of subnets.

LB within a subnet. In this case, the behaviour of the routing agent must be implemented. E.g., in Fig. 2, node N3 belongs to two different subnets. In one subnet, routing agents are of type 1 (e.g. implementing ants based LB algorithm) and in the other one they are of type 2 (e.g. implementing bee based LB algorithm). In order to collaborate with nodes from both subnets, N3 must possess both types of routing agents including both kinds of routing spaces that hold the information specific for each respective LB algorithm. The collaboration between different types of routing agents at N3 goes through its allocation space. It holds the information about partner nodes as computed by the continuously applied location policy. The IN and OUT allocation agents assume that the information about best/suitable partners to/from which to distribute load can be queried any time from the allocation space.

LB between subnets. This level of LB requires a further extended behaviour of routing agents for inter-subnet routing. Note that spaces are represented by XVSM containers that are referenced by URLs. For inter-subnet routing, each routing space is published under a public name using the JXTA based peer-to-peer lookup layer[2] of XVSM so that the routing agents can retrieve the foreign routing spaces in the network. This way, routing within a subnet uses the same pattern as routing between one or more subnets.

[2] http://www.sun.com/software/jxta

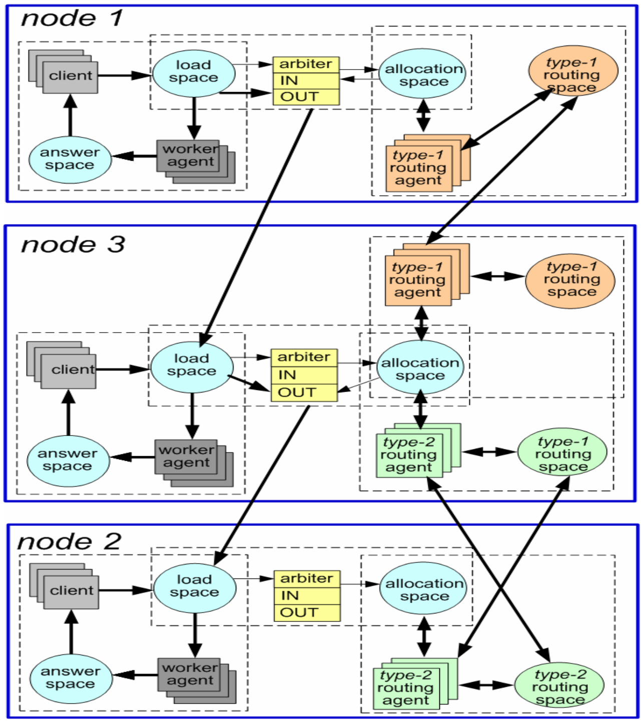

Fig. 2 N1, N2, N3 implementation.

3 Swarm Based Algorithms

The main obstacle in solving the combinatorial optimization problems is that they cannot be solved (optimally) within the polynomial bounded computational time. Therefore, in order to solve large instances, the approximate algorithms (heuristics) have to be used. These algorithms obtain near-optimal solutions in a relatively short time [37]. A set of algorithmic concepts, that can be used to define heuristic methods applicable to a wide set of different problems, was emerged. This new class of algorithms, the so-called metaheuristics, increases the ability of finding very high quality solutions to hard combinatorial optimization problems in a reasonable time [37]. Generally, swarm intelligence describes the collective behavior of fully decentralized, self-organized systems from nature. Particularly successful metaheuristics are inspired by swarm intelligence [10]. This concept belongs to the area of artificial intelligence. Swarm intelligence algorithms refer to a specific set of metaheuristics, adaptive algorithms. Adaptive algorithms usually manipulate with a population of items. Each item is evaluated by means of a figure of merit and its adequacy for the solution. The evaluation is done by using the so-called fitness function. When searching for the adequate solution, exploration and exploitation of a search space are mixed. The exploration investigates unknown areas of the search space, whereas exploitation makes use of accumulated knowledge. A good trade-off between these two contradictory requirements leads to finding a global optimum.

In this section, two types of swarm intelligent algorithms are presented. Bee algorithms are adopted for the LB problem and implemented to this problem for the first time [31]. Although ant algorithms have been applied previously to LB [12], we adapted and implemented them in order for comparison with non-pheromone based swarm intelligence (bees).

Different dynamic processes characterize the LB scenario. Nodes join and leave dynamically, information about load changes permanently, and tasks are dynamically added and continuously processed. A structured peer-to-peer (P2P) network [1] has an overlay topology that is controlled. There is a simple mapping of content to location, and therefore it scales well. On the other side, the support of dynamics is not so good. Queries can only be simple key/value mappings, i.e., exact match queries instead of more complex queries. For these reasons, they are not suitable for the LB problem. In an unstructured P2P network, a placement of information can be done independently of an overlay topology, but the content must be localized explicitly, e.g., through brute force mechanisms or flooding. It is very well suitable for dynamic populations, and complex queries are possible. Therefore, an unstructured P2P network fits better to our problem. The negative point is that it does not scale so well, which is the starting point for improvements. In order to point out the arguments for the potential of using bees for the LB problem, we give a short comparison of Gnutella and swarm-based systems:

Gnutella [1] operates on an unintelligent query flooding based protocol to find a particular node. For communication between peers ping (discover hosts), pong (reply to ping), query (search request), and query hit (reply to query) messages are used. It needs many concurrent agents for one (exhaustive) search, as for each branch, a new agent is required.

Bees search the network and build up routes as overlays. If a bee finds the required information, it directly flies back to its hive and informs the "starting place" of the search directly in a P2P way. Bounding the number of bees is possible, which is an indication that bees can scale better than Gnutella. However, in the first iteration step, there is no guarantee of finding a solution, but one will find a solution in upcoming iterations through learning. Knowledge distribution takes place in the own hive. Bees of different hives do not communicate with each other.

Ants leave information (pheromones) at all nodes on their backward trips. Their forward trip is comparable to the bees' forward movement (navigation), but their backward trip is different as ants do not directly contact the "starting place" in a P2P way but must go the entire way back.

3.1 Bee Algorithm

3.1.1 Bee Behaviour in Nature

A bee colony consists of bees with different roles: foragers, followers, and receivers. This natural intelligence performs self-organization through two main strategies: navigation and recruitment. Navigation means searching for nectar in an unknown landscape. A forager scouts for a flower with good nectar, returns to the hive, unloads nectar, and performs a recruitment strategy, meaning that it communicates the knowledge about the visited flowers to other bees. The best known way of bee communication is the so-called waggle dance which is the main part of the recruitment strategy. Using this “dance language”, a bee informs its hive mates about direction, distance and quality of the food found. A follower randomly chooses to follow a forager and visits the flower that has been “advertised” without own searching. A forager can choose to become a follower in the next step of navigation, and vice versa. A receiver always stays in the hive and processes the nectar. Autonomy, distributed functioning, and self-organization characterize the biological bee behaviour [4].

Bee-inspired algorithms have been applied to several computer science problems like travelling salesman [36], scheduling jobs [6], [28], routing and wavelength assignment in all-optical networks [23], training neural networks for pattern recognition [27], and computer vision and image analysis [26]. Although some of these applications deal with some kind of job scheduling, they differ from our general and domain independent approach as they use a simplified version of a scheduling problem by including the limitations given in advance, e.g., a single machine supplies jobs, and each job needs only one operation to be executed.

3.1.2 Algorithm

In [31], the principals for usage of bee intelligence for LB are proposed. Software agents represent bees at the particular nodes. A node contains exactly one hive and one flower with many nectar units. A task relates to one nectar unit. A hive has a finite number of receiver bees and outgoing (forager and follower) bees. Initially, all outgoing bees are foragers. Foragers scout for a location policy partner node of their node to pull/push nectar from/to it, and recruit followers. The goal is to find the best location policy partner node by taking the best path which is defined to be the shortest one. A suitability function δ (see below) defines the best location policy partner. A navigation strategy determines which node will be visited next and is realized by a state transitions rule [36]:

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

where $\rho_{ij}(t)$ is the arc fitness from node i to node j at time t, 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. In the calculation of the arc fitness values, we differentiate:

(1) **Forager:** A bee behaves in accordance with the state transition rule and $\rho_{ij} = 1/k$, where k is the number of neighbouring nodes of node i. A forager can decide to become a follower in the next cycle of navigation.
(2) **Follower:** Before leaving the hive, bee observes dances performed by other bees and randomly chooses to follow one of the information offered through these dances. This information contains the set of guidance moves that describes the tour from the hive to the destination previously explored by one of its hive mates. This is the so-called *preferred path* [36]. When a bee is at node i at time t, two sets of next visiting nodes can be derived: the set of allowed next nodes, $A_i(t)$ and the set of favoured next nodes, $F_i(t)$. $A_i(t)$ contains the set of neighbouring nodes of node i, whereas $F_i(t)$ contains a single node which is favoured to reach from node i as recommended by the preferred path. The arc fitness is defined in

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

where $|S|$ is the sign of the cardinality (i.e., the number of elements) of some set S. So, $|A_i(t) \cap F_i(t)|$ can be either 0 or 1, as $A_i(t)$ and $F_i(t)$ may have either none element or only one element in their intersection.

A recruitment strategy communicates obtained knowledge about path and quality of solution to bees. From this we can derive a fitness function for bee i,

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

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)

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

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 [25].

Pseudocode1: Bee Colony Optimization (BCO) metaheuristic [36].

```
procedure BCO_MetaHeuristic
  while(not_termination)
        ObserveWaggleDance()
        ConstructSolution()
        PerformWagledance()
  end while
end procedure
```

Each node can start the location policy. If the node is UL, its bee searches for a suitable task belonging to some OL node and carries the information about how complex the task the node can accept. If the node is OL, its bee searches for a UL node that can accept one or more tasks from this OL node. It carries the information about the complexity of tasks this OL node offers and compares it with the available resource of the current UL node that it is visiting. Therefore, the complexity of the task and the available resources at a node must be compared. For this purpose, we need the following definitions: task complexity *c*, host load *hl* and host speed *hs* [8]. *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 $\delta = \delta(x)$ (cf. Table 2). If $x = 1$, then the situation is ideal. The main intention is to find a good location policy partner. For example, when a UL node with high resource capacities is taking work from an OL 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. A detailed description about bee algorithm for LB can be found in [31].

3.1.3 Implementation Parameters

In our implementation, we introduced one parameter, the so-called *search mode* that is configurable and determines which nodes in the network (according to their load status) will trigger a load balancing algorithm.

Table 1 Search Modes.

SM1	the algorithm is triggered from UL nodes, OK nodes (in a situation when it's likely that the node will become OL, but is not yet heavily loaded) and consequently OL nodes
SM2	the algorithm is triggered from UL nodes
SM3	the algorithm is triggered from OK nodes (in a situation when it's likely that the node will become OL, but is not yet heavily loaded) and consequently OL nodes; the computation of x argument for δ(x) suitability is slightly changed[3]
SM4	the algorithm is triggered from OL nodes
SM5	the algorithm is triggered from UL and OL nodes
SM6	the algorithm is triggered from OK nodes (in a situation when it's likely that the node will become OL, but is not yet heavily loaded) and consequently OL nodes.

[3] If a node is in OK state, the algorithm is triggered, and searching for a suitable node among the neighbor nodes is started (afterwards, this information about the most suitable node is stored locally). As soon as the node gets OL, the tasks get re-routed to this target node. To achieve this à priori searching for a suitable node (when the information about a task is still unavailable, i.e., the task complexity c is yet unknown), we compute argument x only on the basis of host speed and host load parameters.

For *suitability* function δ, we implemented the following functions:

Table 2 Suitability Functions.

SF0	one linear function: if (x = 1.0) $\delta(x) = n$, else $\delta(x) = 5x$ (if the number of nodes $\leq n$)
SF1	an exponential function: $\delta(x) = 10^x$
SF2	a polynomial function: $\delta(x) = 10x^3$
SF3	another linear function: if (x < 1.0) $\delta(x) = 4nx$, else $\delta(x) = 5n$ (if the number of nodes $\in [5n-4,5n]$)

The *fitness* function *f* is computed from the suitability function of the found node and the number of hops to this node using the following combinations:

Table 3 Fitness Functions.

FF0	f(x) = δ(x) / number_of_hops
FF1	f(x) = δ(x) • (quality_of_links / number_of_hops)
FF2	f(x) = δ(x) / sqrt(number_of_hops)
FF3	similar to FF0, only the local node is excluded from the comparison and the rest of neighbouring nodes are taken in consideration.

3.2 Ant Algorithms

The basic requirements - to find the *best* location policy partner node by taking the *best* path - are the same as in the bee case. The best location policy partner is defined by the maximum amount of pheromones left on the path. The Ant Colony Optimization metaheuristic (ACO) has been inspired by the real ant colonies. The ants' behaviour is characterized by indirect communication between individuals in a colony via pheromone. A software agent plays the role of an ant. The natural pheromone is stigmergic information that serves as the communication among the agents. Ants make pure local decisions and work in a fully distributed way. In ACO, ants construct solutions by moving from the origin to the destination, step by step, according to a stochastic decision policy. After that, the aim of the pheromone update is to increase the pheromone values associated with good solutions (deposit pheromones) and decrease those associated with bad ones [10].

Pseudocode2: Ant Colony Optimization (ACO) metaheuristic [10].

```
procedure ACO_MetaHeuristic
  while(not_termination)
        ConstructSolutions()
        pheromoneUpdate()
        daemonActions()
  end while
end procedure
```

The most popular variations and extensions of ACO algorithms are: Elitist AS, Rank-Based AS, MinMax Ant System (MMAS), and Ant Colony System. AntNet [9] is a network routing algorithm based on ACO. It is an algorithm for adaptive routing in IP networks, highly adaptive to network and traffic changes, robust to agent failures and provides multipath routing. AntNet algorithm supports adding and removing network components.

The following is a brief "tutorial" about ant algorithms [10], needed for explanation of results and clarification of benchmarks parameters: MinMax [10] is an improvement of the initial Ant System algorithm. In each Ant System algorithm, there are two phases: ants' tour (solution) construction and pheromone update. In the 1st phase, *m* artificial ants concurrently build their solutions starting from randomly chosen nodes and choosing the next node to be visited on their trips by applying a random proportional rule:

$$p_{ij}^k = \frac{[\tau_{ij}]^\alpha [\eta_{ij}]^\beta}{\sum_{l\in N_i^k} [\tau_{il}]^\alpha [\eta_{il}]^\beta}, \text{ if } j \in N_i^k \tag{5}$$

where τ_{ij} is a pheromone trail on (*i,j*)-arc, $\eta_{ij} = 1/d_{ij}$ is a heuristic value (available à priori), α and β are two parameters that determine the influence of the pheromone trail and the heuristic information, and N_i^k is the set of cities that ant *k* has not visited yet. In the 2nd phase, the pheromone trails are updated. The pheromone value on all arcs is decreased by a constant factor:

$$\tau_{ij} \leftarrow (1-\rho)\tau_{ij} \tag{6}$$

where $0 < \rho \leq 1$ is the pheromone evaporation rate. After evaporation, the additional amount of pheromones is deposited on the arcs that have being crossed in the ants' constructions of solutions:

$$\tau_{ij} \leftarrow \tau_{ij} + \sum_{k=1}^{m} \Delta\tau_{ij}^k \tag{7}$$

where $\Delta\tau_{ij}^k$ is the amount of pheromones ant *k* deposits on arcs it has visited.

In the MinMax algorithm, the following modifications are done [10]:

- Best tours found are strongly exploited.
- Possible range of pheromone trail values are limited to the interval $[\tau_{min}, \tau_{max}]$.
- Pheromone trails are initialized to the upper pheromone trail limit.
- Pheromone trails are reinitialized each time the system approaches any kind of stagnation.

So, the 1st phase is the same as in the initial Ant System algorithm, but the 2nd phase is modified – the update of pheromone trails is implemented as follows:

$$\tau_{ij} \leftarrow \tau_{ij} + \Delta\tau_{ij}^{best} \tag{8}$$

where $\Delta\tau_{ij}^{best} = 1/C^{best}$ and C^{best} can be either the length of the iteration's best tour or the length of the best tour so far.

AntNet algorithm [9] is similar to all Ant Algorithms, i.e., has two phases: solution construction and data structures update. The necessary data structures used in this algorithm are: an artificial pheromone matrix τ_i and a statistical model M_i of the traffic situation over the network. Both matrices are associated with each node *i* of the network. Two sets of artificial ants exist: forward ants and backward ants. Generally, ants have the same structure, but their actions differ:

- Forward ant, $F_{s\to d}$, travels from the source node *s* to a destination node *d.*
- Backward ant, $B_{s\to d}$, travels back to the source node *s* using the same path as $F_{s\to d}$ but in the opposite direction; it uses the information collected by $F_{s\to d}$ in order to update routing tables of the visited nodes.

In the 1st phase, each $\mathrm{F}_{s\to d}$ starts its travel from the source node *s* and chooses its destination *d* according to this probabilistic rule:

$$p_{sd} = \frac{f_{sd}}{\sum_{i=1}^{n} f_{si}} \tag{9}$$

where f_{sd} is some measure of data flow. The ant constructs the path on this way:

a) An ant that is currently at node *i* chooses the next node *j* to be visited by applying the following probabilistic rule:

$$P_{ijd} = \frac{\tau_{ijd} + \alpha\eta_{ij}}{1+\alpha(|N_i|-1)} \tag{10}$$

where τ_{ijd} is an element of the pheromone matrix τ_i that indicates the learned desirability for an ant in node *i* with destination *d* to move to node *j*, $|N_i|$ is a number of neighbours of node *i*, η_{ij} is a heuristic value that takes into account the state of the *j*th link queue of the current node *i*:

$$\eta_{ij} = 1 - \frac{q_{ij}}{\sum_{l=1}^{|N_i|} q_{il}} \tag{11}$$

The parameter α from Eq.(10) weighs the importance of the heuristic values with respect to the pheromone values stored in the pheromone matrix.

b) When $F_{s\to d}$ comes to destination node *d*, it generates $B_{s\to d}$, transfers to it all of its memory and is deleted.

c) $B_{s\to d}$ travels back to the source node *s* using the same path as $F_{s\to d}$ but in the opposite direction. It uses the information collected by $F_{s\to d}$ in order to update the routing tables of the visited nodes.

The 2nd phase considers updating matrices τ_i and M_i. In the pheromone matrix τ_i, values that suggest choosing neighbour *f* when destination is *d*, are incremented:

$$\tau_{ifd} \leftarrow \tau_{ifd} + r\cdot(1-\tau_{ifd}) \tag{12}$$

The other pheromone values are decremented:

$$\tau_{ijd} \leftarrow \tau_{ijd} - r\cdot\tau_{ijd} \quad j\in N_i, \;\; ; j\neq f \tag{13}$$

There are several ways to determine and assign r values: from the simplest way of setting r = *constant* to a more complex way that defines r as a function of the ant's trip time and the parameters of the statistical model M_i.

Remodelling of these ant algorithms for a location policy comprises the following changes. What does "Construct Solution" mean in our case? The ant made a path and found the data on that path. We are not only interested in the best path, but also in the quality of the data found. Therefore, *DepositPheromone* procedure is changed as follows. If an ant on its trip:

1. Found exact data, it deposits pheromone;
2. Found acceptable data with the accuracy/error rate < ε, (ε is a parameter given in advance related to the definition of δ), it deposits less amount of pheromone,
3. Did not find data, then skips depositing pheromones on its trip (i.e., the values on arcs it traversed will be the same as the values on the rest of unvisited arcs in the network).

A different amount of pheromones is deposited according to the quality of the solution found. The suitability function $\delta = \delta(x)$ describes how good (acceptable) the found solution is, $\delta \in [0,1]$. In case of changing the type of δ, its value can be scaled into the same segment [0,1]. DepositPheromone procedure is changed:

1. For MinMax algorithm: $\Delta\tau = 1/MC^{best}$ where $M=1/\delta$;
2. For AntNet algorithm: $\tau := r\cdot(1-\tau)\cdot\delta$.

4 Benchmarks

This section describes the benchmarks performed in the SILBA framework. As a detailed explanation of the basic SILBA benchmarks can be seen in [32], emphasis is put on more sophisticated benchmarks in the extended SILBA. Therefore, the basic SILBA benchmarks, i.e., their conclusion are mentioned briefly here.

4.1 Basic SILBA Benchmarks

The tests are constructed on the basis of the following criterions [32]:

- Find out the best combination of parameter settings for each intelligent algorithm: Bee Algorithm, MinMax and AntNet Ant Algorithms,
- Compare these optimally tuned swarm based algorithms with several well-known algorithms: Round Robin, Sender, Adapted Genetic Algorithm (GA).

The benchmarks demonstrate: the agility of the SILBA pattern by showing that algorithms can be easily exchanged, and the promising approach of bee algorithms.

The load is generated by one single client. For performing test examples, an arbitrary topology is used in which a full connection between all nodes is not required. All benchmarks are carried out on a cluster of 4 machines, and on the Amazon EC2 Cloud. As the figure of merit, the absolute execution time and scalability of the solutions are used. The values of the suitability function help to discern the usefulness of intelligent algorithms and emphasize the correctness of properly chosen partner nodes, i.e., the methodology to determine the best partner node. This function reflects how a good solution is chosen and the degree of self-organization of the used swarms. The average *x* value is 1, meaning the best node is always chosen.

In the basic SILBA, the best combination of feasible parameters for each algorithm is identified and, under these conditions, the advantages of using bee swarm intelligence in the context of load balancing are presented. The obtained results show that the bee algorithm behaves well, does not impose an additional complexity and outperforms all other test candidates [32].

4.2 Extended SILBA Benchmarks

As the extended SILBA supports the multi-level LB strategy, the goal was to exchange the algorithms on each level. In the considered case, there are 2 levels on which LB is realized concurrently: between several subnets and inside each subnet. Also, the success of a particular combination depends on a network topology. The tests are performed on the basis of the following criterions:

- Find the best combination of algorithms for each of the well-known topologies (chain, full, ring, star) used in these benchmarks.
- Compare and analyze the best obtained combinations.
- After obtaining the best combinations, perform the benchmarks on different network (and subnets) dimensions and evaluate the scalability issue.

The benchmarks demonstrate: 1) the flexibility of the SILBA pattern by showing that the LB problem could be easily treated in more complex network structures with several subnets, 2) detection of those topologies which profit most of swarm intelligent algorithms (particularly bee algorithms).

4.2.1 Test Examples and Test Environment

Test examples are constructed taking into account the following issues: the combination of algorithms, the different number of subnets and the number of nodes per subnets, the increased number of clients per each subnet, different topologies:

Combinations (36) of all algorithms on two levels (6 algorithms on 2 levels): Level 1 denotes the used algorithm inside a subnet, whereas level 2 denotes the used algorithm between subnets; the *values* of the respective parameters are described in Table 4 and reused from basic SILBA.

Table 4 Combinations of algorithms.

level1 / level2	Bee Alg.	MinMax	AntNet	adaptedGA	Sender	Round Robin
Bee Alg.	1	2	3	4	5	6
MinMax	7	8	9	10	11	12
AntNet	13	14	15	16	17	18
adaptedGA	19	20	21	22	23	24
Sender	25	26	27	28	29	30
Round Robin	31	32	33	34	35	36

Different number of subnets and number of nodes per subnets:

Table 5 Distribution of nodes in subnets.

total number of nodes	number of subnets	number of nodes in each subnet
16	4	4
16	8	2
32	4	8
32	8	4

Increased number of clients per each subnet:
In the basic SILBA, only one client is responsible for putting the tasks into the network. This leads to a light to moderate loaded network. In extended SILBA, the number of clients is drastically increased, i.e., for a subnet of *n* nodes, the assigned number of clients is *n/2*. The number of clients per subnet is increased until the subnet becomes fully loaded. Each client supplies the same number of tasks. Clients are symmetrically positioned in order to have fairly loaded subnet. The same parameter is used for all test runs.

Different topologies:
The well-known topologies, ring, star, full, chain, are chosen in order to define which combination of algorithms fits the best to a particular topology. Fig. 3 depicts one example of each topology. Subnets can be with intersections and without intersections, but in both cases at least one node from each subnet must possess two types of routing agents in order to allow for the realization of different types of load balancing algorithms (inside a subnet, between subnets).

Two different *test environments* are used: a cluster of 4 machines, and the Amazon EC2 Cloud[4]. Each machine of the cluster had the following characteristics:

[4] http://aws.amazon.com/ec2/

2*Quad AMD 2,0GHz with 16 GB RAM. We simulated a network with 16 (virtual) nodes. Each test run began with a “cold start" and all nodes were being UL. On Amazon Cloud, we used standard instances of 1.7 GB of memory, 1 EC2 Compute Unit (1 virtual core with 1 EC2 Compute Unit), 160 GB of local instance storage, and the 32-bit platform.

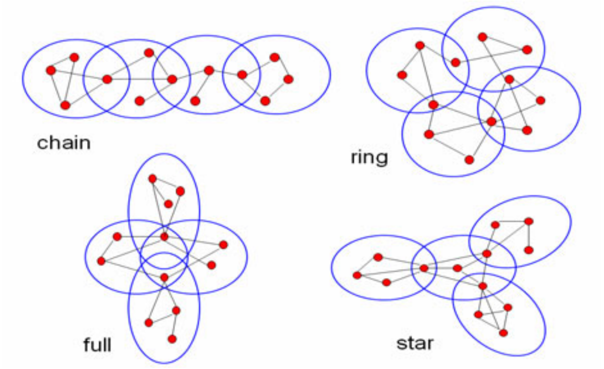

Fig. 3 Topology examples.

4.2.2 Raw Result Data

The next figures (Fig. 4 – Fig. 7) show all combinations of algorithms on different topologies, searching for the best combination in each topology. The presented results demonstrate a 4*4 structure, i.e., 4 subnets and 4 nodes in each subnet. In each subnet, each client supplies 200 tasks, giving a total of 1600 tasks.

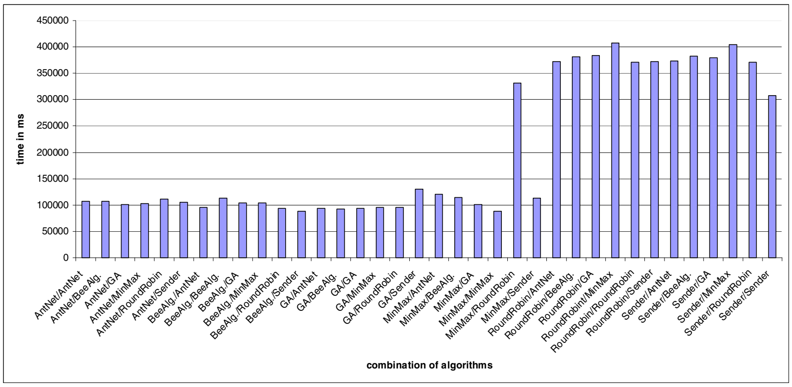

Fig. 4 Combination of algorithms in chain topology.

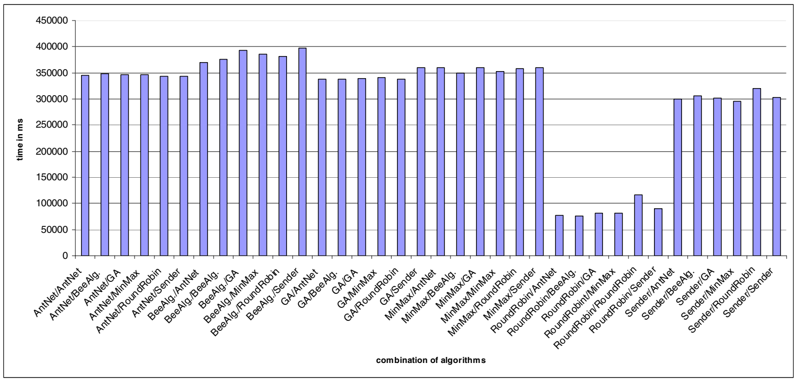

Fig. 5 Combination of algorithms in full topology.

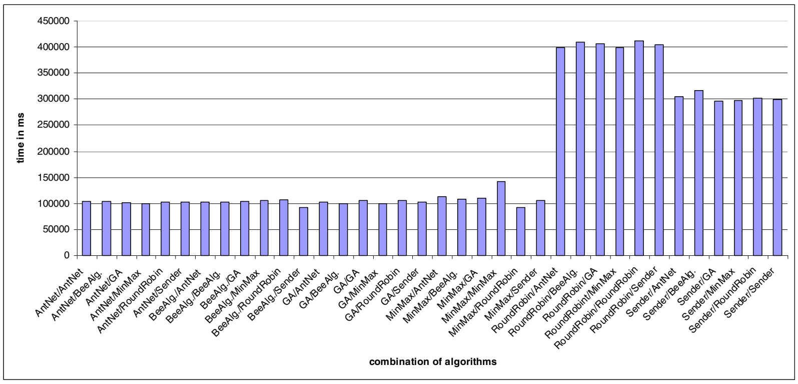

Fig. 6 Combination of algorithms in ring topology.

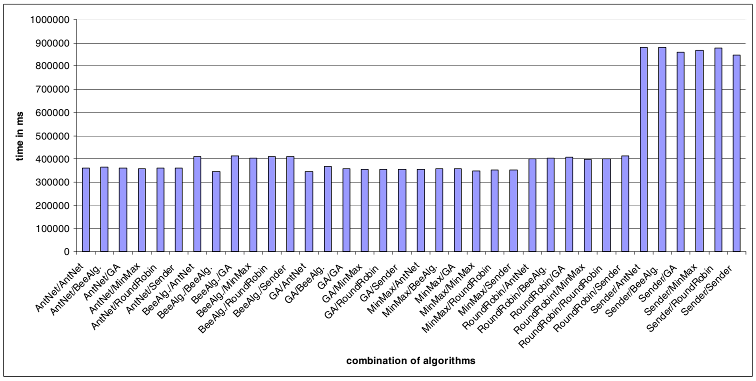

Fig. 7 Combination of algorithms in star topology.

On the basis of the results obtained (Fig. 4 – Fig. 7), the overall comparison is done (Table 6). Many appearances of the same topology in Table 6 denote that the respective combinations are equally good (e.g., both combinations Bee Alg./Sender and MinMax/MinMax are equally good in a chain topology).

As can be noticed, in each topology (except star) the best combination is made by one intelligent and one unintelligent algorithm. Although these combinations are not real hybrid algorithms (each pure algorithm works either inside a subnet or between subnets), the overall load distribution in the entire network is realized through their synergy. Intelligent algorithms find good starting solutions (quality and fastness), while unintelligent algorithms improve these solutions (fastness).

Table 6 Overall comparison of the best results in all topologies.

topology	combination of algor.	time (ms)
chain	BeeAlg./Sender	88000
chain	MinMax/MinMax	88000
full	RoundRobin/BeeAlg.	76000
ring	BeeAlg./Sender	93000
ring	MinMax/RoundRobin	93000
star	BeeAlg./BeeAlg.	346000
star	GA/AntNet	346000

The results from Table6 are graphically presented in Fig.8.

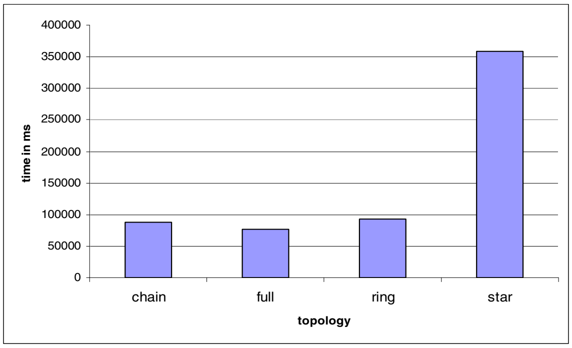

Fig. 8 Results of the best combinations for each topology.

After obtaining the best combination for each topology, the benchmarks with the best combinations are performed on larger network dimensions. Table7 summarizes these results and shows that the results are stable as the same combination(s) of algorithms are obtained as the best ones for each of different dimensions (4*4, 8*2, 4*8, 8*4).

Table 7 Results (time in ms) of the best combinations in different network dimensions.

total number of nodes	number of subnets	number of nodes in each subnet	chain	full	ring	star
16	4	4	88000	76000	93000	346000
	8	2	374000	384000	359000	365000
32	4	8	420000	556000	582000	388000
	8	4	406000	455000	484000	356000

Extended SILBA offers better and more powerful solutions than basic SILBA. The situations that can benefit from extended SILBA are the following:

1. Subnets are physically required.
2. Extremely large networks with a high number of nodes where building subnets and applying the extended SILBA strategy helps transferring the load between very distant nodes. Load need not be transferred via a number of hops from one node to another one, but can be transferred by using a shortcut, "jumping" from the original node's subnet to the distant destination node's subnet.

4.2.3 Overall Evaluation

The *absolute execution time* is used as metric for the benchmarks. According to the obtained results (see section 4.2.2.), the behaviour of a particular combination of algorithms depends on a topology. The questions to be analyzed are:

1. How much is the best combination (in each topology) better than "extreme" combinations: the worst one and the combination of the second best one?
2. What is the "behaviour" of the other combinations, i.e., how much do they deviate from the best solution? What is the "collective behaviour" of algorithm combinations and the used SILBA pattern in each topology?

For *chain* topology, the best result is obtained by both BeeAlgorithm/Sender and MinMax/ MinMax. They are equally good, and 5.4% better than the combination in the second place, GA/Bee Algorithm, 78% better than the worst combination, and 56% better than the average of all combinations. The additional measurements, the interval of variation and the root mean square deviation (RMSD) are introduced in order to examine the behaviour of the other combinations, i.e., how much they deviate from the best solution. The interval of variation is defined as the difference between the maximum value of the used metric (time) and its minimum value: $t_{max} – t_{max}$ and is equal to 320000ms. The used RMSD is a quantitative measure (a decimal number) that tells how many good combinations in a particular topology exist, i.e., how far from the best solution the data points (the rest of the combinations) tend to be (smaller RMSD means more good combinations). For chain topology, the value of RMSD is 172121.

The combination RoundRobin/BeeAlgorithm shows the best results in the *full* topology. This combination is 1.3% better than the combination in the second place, RoundRobin/AntNet, 80.9% better than the worst combination, and 74.9% better that the average of all combinations. The interval of variation, $t_{max} - t_{max}$, is 322000ms and the RMSD is 248227.7.

Both BeeAlgorithm/Sender and MinMax /RoundRobin are equally good in the *ring* topology. They are 1.4% better than the combination in the second place, MinMax/RoundRobin, 60.7% better than the worst combination, and 24.3% better that the average of all combinations. The interval of variation, $t_{max} - t_{max}$, is 535000ms and the RMSD is 216194.9.

For the *star* topology, the combinations BeeAlgorithm/BeeAlgorithm and GA/AntNet are the best with the same resulting value. They are 6.1% better than the combination in the second place, AntNet/MinMax, 77.4% better than the worst combination, and 50.1% better that the average of all combinations. The interval of variation, $t_{max} - t_{max}$, is 319000ms and the RMSD is 153859.9.

From these results we can conclude that bee algorithms play an important role in almost every topology. The best obtained results in each topology are based on bee algorithms used either inside or between subnets, or in both. Also, the rest of intelligent algorithms give good results in all topologies. The exception is the full topology where the best results are obtained when round robin algorithm is used inside subnets and combined with all others algorithms (except the combination Round Robin/Round Robin).

The RMSD shows that the greatest deviation is reached in full topology, i.e., the majority of the other combinations differentiate a lot (they are worse in a significant extent) comparing to the best obtained combination. The smallest deviation is in star topology, so the combinations behave evenly in this topology. If we analyze how good response will be obtained by plugging any (random) combination of algorithms into SILBA, the equally good results are obtained in star topology. So, SILBA is very stabile (without peaks in results) in star topology. At the other side, Fig.8 shows that the results of the individual combinations of the SILBA pattern are successful for chain, full and ring topologies, whereas the results obtained for star topology are not so good.

In the next table, the behaviour of the swarm intelligence algorithms' combinations is extracted as these algorithms are promising ones and not exploited so much. Table8 shows how much they deviate from the best solution in each of the used topologies. For example, the set of all combinations that use bee algorithms inside subnets is denoted in the table as "bee/others". According to these results, all the combinations from this set deviate slightly from the best combination in the chain topology (that are BeeAlgorithm/Sender and MinMax/MinMax), whereas the combinations from this set deviate more from the best combination in star topology, although the best combination is BeeAlgorithm/BeeAlgorithm.

Table 8 Deviation swarm based algorithms' combinations from the best solution.

	chain	full	ring	star
RMSD (Bee/Others)	35171.0	755211.9	25337.7	141470.8
RMSD (Others/Bee)	417868.4	600503.1	387401.6	537938.7
RMSD(AntNet/Others)	44899.9	659335.3	25869.2	35787.1
RMSD(Others/AntNet)	404891.3	608559.9	372385.6	541636.4
RMSD(MinMax/Others)	249164.6	686738.7	58813.3	21725.6
RMSD(Others/MinMax)	450334.3	603189.0	371052.6	526899.4

Additionally, scalability is analyzed. Here, we focus on the issue of *load scalability*. A very general definition of scalability is taken into account [14], [35]. According to [14], a general family of metrics can be based on the following definition:

$$\psi = \frac{F(\lambda_2, QoS_2, C_2)}{F(\lambda_1, QoS_1, C_1)} \tag{14}$$

where F evaluates the performance, λ evaluates the rate of providing services to users, QoS is a set of parameters which evaluate the quality of the service seen by users, and C reflects the cost. Further, [14] establishes the scaling strategy by means of a scaling factor k and a set of scaling variables which are functions of k. They express the strategy as a scaling path in a space in which they are the coordinates. Fig.9 shows how $\psi(k)$ behaves in different situations:

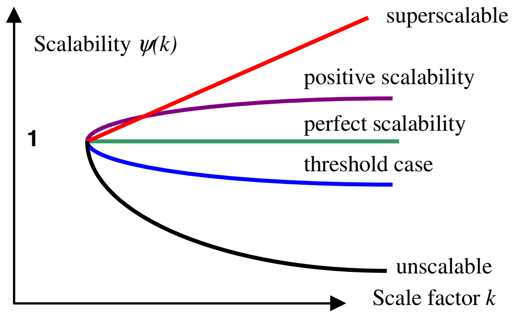

Fig. 9 Scaling behavior [14].

We specialize it to a simplified version of interest to our problem in terms of load, resources and performance measure. This restricted aspect of scalability can be quantitatively described on the basis of the computational resources available (R), load of the system (L) and some performance measure (P). Then scalability can be quantified by means of a "scalability ratio" r_{scal} for a given constant k

$$r_{scal} = \frac{P(L,R)}{P(kL,kR)} \tag{15}$$

Usually, performance *P* is the function of load *L* and resources *R*. A certain aspect of scalability is described by the answer to the question of how *P* is affected when more resources (larger *R*) have to compensate for more load (larger *L*). A constant remaining value of *P* when simultaneously increasing *L* and *R* by the same factor leads to the "ideal" scalability ratio of 1. In our test examples, this interpretation of load scalability is applied. We analyze the increasing of load with the increasing of the resources. By comparing results (Table 12), it is easy to see that the best chosen combinations based on bee algorithm scale well [14]. Load and resources are increased *twice* for consecutive test runs.

Scalability in basic SILBA

For example in the cluster environment, load and resources are increased twice for consecutive test runs, i.e., they are increased by 2^n compared with the starting test run (4 nodes, 50 tasks). The values of r_{scal} are 2.9, 3.0, 3.4, 3.2 (rounded to one decimal) for consecutive bee test runs, i.e., 2.9, 9.1, 31.0, 100.8 compared with the starting test run (4 nodes, 50 tasks). These values converge to positive scalability. Such behaviour is even better in a more real environment, i.e., on the Cloud. Almost the similar situation occurs with AntNet algorithm.

Scalability in extended SILBA

For *chain* topology: a) If the number of subnets is increased and the number of nodes inside a subnet is the same, i.e., **4***4, **8***4, r_{scal} is 4.6 (rounded to one decimal), that leads to positive scalability. b) If the number of nodes in a subnet is increased and the number of subnets is the same, i.e., 4***4**, 4***8**, r_{scal} is 4.8; 8***2**, 8***4**, r_{scal} is 1.1, that converges to perfect scalability.

For *full* topology: a) If the number of subnets is increased and the number of nodes inside a subnet is the same, i.e., **4***4, **8***4, r_{scal} is 5.98; that leads to positive scalability. b) If the number of nodes in a subnet is increased and the number of subnets is the same, i.e., 4***4**, 4***8**, r_{scal} is 7.3; 8***2**, 8***4**, r_{scal} is 1.8; that leads to positive scalability.

For *ring* topology: a) If the number of subnets is increased and the number of nodes inside a subnet is the same, i.e., **4***4, **8***4, r_{scal} is 5.2; that leads to positive scalability. b) If the number of nodes in a subnet is increased and the number of subnets is the same, i.e., 4***4**, 4***8**, r_{scal} is 6.2; 8***2**, 8***4**, r_{scal} is 1.3; that converges to perfect scalability.

For *star* topology: a) If the number of subnets is increased and the number of nodes inside a subnet is the same, i.e., **4***4, **8***4, r_{scal} is approximately 1; that leads to perfect scalability. b) If the number of nodes in a subnet is increased and the number of subnets is the same, i.e., 4***4**, 4***8**, r_{scal} is approximately 1; 8***2**, 8***4**, r_{scal} is approximately 1; that leads to perfect scalability.

5 Conclusion

In this chapter, the problem of dynamic load balancing is investigated and treated. First, the generic load balancing architectural pattern SILBA is introduced and shortly explained. It allows the plugging and easy exchanging of a variety of algorithms. First, SILBA is developed in its basic form, which refers to load balancing within one network. Later, SILBA is extended in a way that load balancing can be done through different levels (between nodes in one network, between subnets in one network, between several networks) and this can be done concurrently. Different load balancing algorithms can be plugged into SILBA.

In the basic SILBA, the advantages of using bee swarm intelligence in the context of load balancing are presented. Besides bee swarm intelligence, two further intelligent algorithms are adapted based on MinMax and AntNet ant algorithms. For these algorithms, the best combination of feasible parameters is identified, and they are compared with three well-known algorithms: Round Robin, Sender, and Adapted Genetic Algorithm. The load is generated by one single client, and as a performance parameter the absolute execution time is used. Under these conditions, the obtained results show that the bee algorithm outperforms all other test candidates. All benchmarks are carried out on a cluster of 4 machines, and on the Amazon EC2 Cloud.

The extended SILBA shows the advantages of using bee swarm intelligence in the combination with the other algorithms (both intelligent and unintelligent) for load balancing in more complex network structures that consist of different subnets which might overlap or be nested: investigating different network topologies, the combinations that are based on swarm algorithms show the best results in the chain, ring and full topologies. The best combinations in all topologies are based on bee algorithms. The load is generated by many clients, positioned symmetrically in subnets. The benchmarks are also carried out on a cluster of 4 machines, and on the Amazon EC2 Cloud. The performance measure is the absolute execution time, expressed in milliseconds. The best obtained combinations scale well in all investigated topologies.

Future work will concern the following issues:

- Except execution time and scalability, different metrics will be used for the evaluation of results and analysis: communication delay, utilization, stability, fairness across multi-user workloads, robustness in the face of node failure, adaptability in the face of different workloads, etc. Also, it will be investigated under which circumstances, each metric is most appropriate.
- Benchmarking of very large instances and specific examples of the state of the art in the real world.
- Investigation of the impact of load injection in different places in the network.
- Although enlarging their parameter space is the part of the future work, the other way of investigation will play a role, i.e., a shrinking of the parameter space, with more samples and more determinism so that the nature of swarm intelligent algorithms (especially bee intelligence) can be better understood.

- Developing of a recommendation system for a given problem, e.g., the determination of the best topology, algorithm combinations, and parameters tuning for a particular problem.
- Application of the results to distribute load in collaborative security scenarios like distributed spam analysis and intrusion detection.

Acknowledgments. The work is partially funded by the Austrian Government under the program FIT-IT (Forschung, Innovation und Technologie für Informationstechnologien), project 825750 Secure Space - A Secure Space for Collaborative Security Services. We would also like to thank Deguang Sea and Fabian Fischer for implementation and benchmarking of SILBA.

References

[1] Androutsellis-Theotokis, S., Spinellis, D.: A survey of peer-to-peer content distribution technologies. ACM Computing Surveys 36(4), 335–371 (2004)

[2] Bronevich, A.G., Meyer, W.: Load balancing algorithms based on gradient methods and their analysis through algebraic graph theory. Journal of Parallel and Distributed Computing 68(2), 209–220 (2008)

[3] Cabri, G., Leonardi, L., Zambonelli, F.: Mars: A programmable coordination architecture for mobile agents. IEEE Internet Computing 4(4), 26–35 (2000)

[4] Camazine, S., Sneyd, J.: A model of collective nectar source selection by honey bees: Self-organization through simple rules. Journal of Theoretical Biology 149, 547–571 (1991)

[5] Chen, J.C., Liao, G.X., Hsie, J.S., Liao, C.H.: A study of the contribution made by evolutionary learning on dynamic load-balancing problems in distributed computing systems. Expert Systems with Applications 34(1), 357–365 (2008)

[6] Chong, C.S., Sivakumar, A.I., Low, M.Y., Gay, K.L.: A bee colony optimization algorithm to job shop scheduling. In: Proceedings of the Thirty-Eight Conference on Winter Simulation, pp. 1954–1961 (2006)

[7] Cortes, A., Ripolli, A., Cedo, F., Senar, M.A., Luque, E.: An asynchronous and iterative load balancing algorithm for discrete load model. Journal of Parallel and Distributed Computing 62(12), 1729–1746 (2002)

[8] Da Silva, D.P., Cirne, W., Brasileiro, F.V.: Trading Cycles for Information: Using Replication to Schedule Bag-of-Tasks, pp. 169–180. Applications on Computational Grids, Proceeding of European Conference on Parallel Processing (2003)

[9] Di Caro, G., Dorigo, M.: AntNet: Distributed Stigmergetic Control for Communications Networks. Journal of Artificial Intelligence Research 9, 317–365 (1998)

[10] Dorigo, M., Stuetzle, T.: Ant Colony Optimization. MIT Press, Cambridge (2005)

[11] Gelernter, D., Carriero, N.: Coordination languages and their significance. ACM Communication 35(2), 97–107 (1992)

[12] Ho, C., Ewe, H.: Ant colony optimization approaches for the dynamic load-balanced clustering problem in ad hoc networks. In: Proceeding of Swarm Intelligence Symposium, IEEE/SIS 2007, pp. 76–83 (2007)

[13] Janssens, N., Steegmans, E., Holvoet, T., Verbaeten, P.: An agent design method promoting separation between computation and coordination. In: Proceedings of the 2004 ACM Symposium on Applied Computing, SAC 2004, pp. 456–461 (2004)

[14] Jogalekar, P., Woodside, C.M.: Evaluating the Scalability of Distributed Systems. IEEE Transanctions on Parallel and Distributed Systems 11(6), 589–603 (2000)
[15] Kühn, E., Mordinyi, R., Schreiber, C.: An extensible space-based coordination approach for modelling complex patterns in large systems. In: Proceedings of the Third International Symposium on Leveraging Applications of Formal Methods, pp. 634–648 (2008)
[16] Kühn, E., Riemer, J., Lechner, L.: Integration of XVSM spaces with the Web to meet the challenging interaction demands in pervasive scenarios. Ubiquitous Computing and Communication Journal - Special issue of Coordination in Pervasive Environments 3 (2008)
[17] Kühn, E., Mordinyi, R., Keszthelyi, L., Schreiber, C.: Introducing the Concept of Customizable Structured Spaces for Agent Coordination in the Production Automation Domain. In: Proceedings of the Eighth International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2009, pp. 625–632 (2009)
[18] Kühn, E., Mordinyi, R., Lang, M., Selimovic, A.: Towards Zero-delay Recovery of Agents in Production Automation Systems. In: Proceedings of the 2009 IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology, vol. 2, pp. 307–310 (2009)
[19] Kühn, E., Sesum-Cavic, V.: A space-based generic pattern for self-initiative load balancing agents. In: Aldewereld, H., Dignum, V., Picard, G. (eds.) ESAW 2009. LNCS (LNAI), vol. 5881, pp. 17–32. Springer, Heidelberg (2009)
[20] Kühn, E.: Virtual Shared Memory for Distributed Architectures. Nova Science Publishers (2001)
[21] Lemmens, N., De Jong, S., Tuyls, K., Nowé, A.: Bee behaviour in multi-agent systems. In: Tuyls, K., Nowe, A., Guessoum, Z., Kudenko, D. (eds.) ALAMAS 2005, ALAMAS 2006, and ALAMAS 2007. LNCS (LNAI), vol. 4865, pp. 145–156. Springer, Heidelberg (2008)
[22] Lin, F.C., Keller, R.M.: The gradient model load balancing method. IEEE Transactions On Software Engineering 13(1), 32–38 (1987)
[23] Markovic, G., Teodorovic, D., Acimovic-Raspopovic, V.: Routing and wavelength assignment in all-optical networks based on the bee colony optimization. AI Communications 20(4), 273–285 (2007)
[24] Mordinyi, R., Kühn, E., Schatten, A.: Towards an Architectural Framework for Agile Software Development. In: Proceedings of the Seventeenth International Conference and Workshops on the Engineering of Computer-Based Systems, pp. 276–280 (2010)
[25] Nakrani, S., Tovey, C.: On honey bees and dynamic server allocation in the Internet hosting centers. Adaptive Behaviour 12(3-4), 223–240 (2004)
[26] Olague, G., Puente, C.: The Honeybee Search Algorithm for Three-Dimensional Reconstruction. In: Rothlauf, F., Branke, J., Cagnoni, S., Costa, E., Cotta, C., Drechsler, R., Lutton, E., Machado, P., Moore, J.H., Romero, J., Smith, G.D., Squillero, G., Takagi, H. (eds.) EvoWorkshops 2006. LNCS, vol. 3907, pp. 427–437. Springer, Heidelberg (2006)
[27] Pham, D.T., Soroka, A.J., Ghanbarzadeh, A., Koç, E., Otri, S., Packianather, M.: Optimising neural networks for identification of wood defects using the Bees Algorithm. In: Proceedings of the IEEE International Conference on Industrial Informatics, pp. 1346–1351 (2006)
[28] Pham, D.T., Koç, E., Lee, J.Y., Phrueksanant, J.: Using the Bees Algorithm to schedule jobs for a machine. In: Proceedings of the Eighth International Conference on Laser Metrology, pp. 430–439 (2007)

[29] Picco, G.P., Balzarotti, D., Costa, P.: Lights: a lightweight, customizable tuple space supporting context-aware applications. In: Proceedings of the ACM Symposium on Applied Computing, SAC 2005, pp. 413–419 (2005)
[30] Picco, G.P., Murphy, A.L., Roman, G.C.: Lime: Linda meets mobility. In: Proceedings of the IEEE International Conference on Software Engineering, pp. 368–377 (1999)
[31] Šešum-Čavić, V., Kühn, E.: Instantiation of a generic model for load balancing with intelligent algorithms. In: Hummel, K.A., Sterbenz, J.P.G. (eds.) IWSOS 2008. LNCS, vol. 5343, pp. 311–317. Springer, Heidelberg (2008)
[32] Šešum-Čavić, V., Kühn, E.: Comparing configurable parameters of Swarm Intelligence Algorithms for Dynamic Load Balancing. In: Proceedings of the Fourth IEEE International Conference on Self-Adaptive and Self-Organizing Systems, Workshop Self-Adaptive Network, SASO/SAN, pp. 255–256 (2010)
[33] Shivaratri, N.G., Krueger, P.: Adaptive Location Policies for Global Scheduling. IEEE Transactions on Software Engineering 20, 432–444 (1994)
[34] Shoham, Y., Leyton-Brown, K.: Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations. Cambridge University Press, Cambridge (2009)
[35] Van Steen, M., Van der Zijden, S., Sips, H.J.: Software Engineering for Scalable Distributed Applications. In: Proceedings of the Twenty-Second International Computer Software and Applications Conference, COMPSAC, pp. 285–293 (1998)
[36] Wong, L.P., Low, M.Y., Chong, C.S.: A Bee Colony Optimization for Traveling Salesman Problem. In: Proceedings of the Second Asia International Conference on Modelling & Simulation, AMS, pp. 818–823. IEEE, Los Alamitos (2008)
[37] Yang, X.: Nature-Inspired Metaheuristic Algorithms. Luniver Press (2008)
[38] Zhou, S.: A trace-driven simulation study of dynamic load balancing. IEEE Transactions on Software Engineering 14(9), 1327–1341 (1988)

Glossary

FF	fitness function
GA	genetic algorithm
LB	load balancing
LP	location policy
MINMAX	min-max ant system algorithm
OK	ok-loaded
OL	overloaded
SF	suitability function
SILBA	self initiative load balancing agents
SM	search mode
TP	transfer policy
UL	under-loaded
XVSM	extensible virtual shared memory