A Space-Based Generic Pattern for Self-Initiative Load Balancing Agents

Eva Kühn and Vesna Sesum-Cavic

Vienna University of Technology, Institute for Computer Languages,
Space Based Computing Group, Argentinierstr. 4, 1040, Wien, Austria
{eva,vesna}@complang.tuwien.ac.at

Abstract. Load-Balancing is a significant problem in heterogeneous distributed systems. There exist many load balancing algorithms, however, most approaches are very problem specific oriented and a comparison is therefore complex. This paper proposes a generic architectural pattern for a load balancing framework that allows for the plugging of different load balancing algorithms, reaching from unintelligent to intelligent ones, to ease the selection of the best algorithm for a certain problem scenario. As in complex network environments there is no "one-fits-all solution", also the integration of several different algorithms shall be supported. The presented pattern assumes autonomous agents and decentralized control. It can be composed towards arbitrary network topologies, foresees exchangeable policies for load-balancing, and uses a black-board based communication mechanism to achieve high software architecture agility. The pattern has been implemented and first instantiations of it with three algorithms have been benchmarked.

Key words: Load balancing, self-organization, autonomous agents, coordination patterns, intelligent algorithms, complex distributed systems.

1 Introduction

The rapid growth of computer systems and their complexity imposes the necessity to reconsider dynamic load balancing (LB) in order to improve the performance of the overall distributed system and to achieve the highest level of productivity. 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. LB 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. At the local level, the determining factor for load distribution is the balanced utilization of all core processors. At the network level, 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 problem becomes even more complex in heterogeneous systems. Networks are growing constantly and therefore the intensive need of including self-* properties (self-organization, self-management, self-repairing, self-configuring, self-grouping, self-learning, self-adaptation, etc.) arise to deal with increasing complexity. We can find a wide range of LB approaches (see section 2), however, our objections address the lack of: Provisioning a general framework, autonomy and self-* properties, and arbitrary configurations. *Provisioning of a General Framework:* Existing LB approaches are very problem specifically oriented (see section 2). As there is no "one-fits-all solution, in order to find a best solution for a problem, a generalized framework is needed that allows for testing and tuning different LB algorithms for a specific problem and environment. The framework shall support easy and dynamic exchange of algorithms as well as combinations of different algorithms. The architecture shall be agile, so that neither new requirements on LB algorithms, nor other assumptions on the network infrastructure, nor the dynamic joining and leaving of agents do become "architecture breakers". Note that a framework itself doesn't solve the LB problem but serves as a necessary basement for LB algorithms. It abstracts the general requirements in a flexible software architecture. *Autonomy and Self-* Properties:* Increased complexity of software systems, diversity of requirements, and dynamically changing configurations, force to find new solutions 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.

In this paper, we present an extensible coordination pattern called SILBA (Self-Initiative Load Balancing Agents) with pluggable LB algorithms and autonomic (multi) agents. The main contribution of this paper is the design of a reusable and agile architectural pattern and its independence on the problem at hand. Section 2 gives a classification of existing LB algorithms. Section 2 describes the SILBA pattern and how it can be extended towards arbitrary network configurations. Section 3 presents the implementation of a LB framework based on the SILBA pattern, using a space-based middleware. In section 5, we choose some well-known algorithms (both unintelligent and intelligent) as examples, map them to SILBA, and show some benchmarks. In section 6 we summarize the results and further research work.

2 Classification of LB Algorithms

There are many different approaches that cope with the LB problem. As a first classification, we shortly listen the most important ones and classify them according to the underlying LB algorithm:

The *first* group consists of different conventional approaches without using any kind of intelligence, e.g.: Sender Initiated Negotiation and Receiver Initiated Negotiation [27], Gradient Model [30], Random Algorithm [14], and Diffusion Algorithm [9]. In Sender algorithm, LB is initiated by an overloaded node. This algorithm has a good performance for low to moderate load level while in Receiver algorithm, LB is initiated by an under-loaded node and this algorithm has a good performance for moderate to heavy load level. Also the combination of these two algorithms (Symmetric) is possible. 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 certain time period, it sends the newly arrived task to a randomly chosen node without taking in consideration whether the target node is overloaded 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 [5] without focus on implementation and benchmarks. The *third* group contains approaches that use intelligent algorithms like evolutionary approaches [7], and ant colony optimization approaches ([20], [41]). Evolutionary approaches use the adjustment of some parameters specific for evolutionary algorithms to achieve the goal of LB. Ant colony optimization approach is used in [20] for a graph theoretic problem formulated from the task of computing load balanced clusters in ad hoc networks. These approaches mainly try to improve only one of the components of the whole LB infrastructure, namely the LB algorithm. In this paper we will propose a framework that allows for the integration of different kinds of algorithms. However, a particular focus is on autonomous agents based control. There exist interesting approaches for solving the LB problem by using agent-based models. According to multi-agent systems, LB can be either static or mobile [18]. In static LB, tasks cannot be migrated elsewhere once they have been launched on a specific server. In mobile LB, a task may migrate to another server, utilizing the agent's mobility. In [26], it is shown that mobile LB outperforms the static case with a 3-40 % improvement over the static placement scheme.

	framework abstraction	**no framework**
agents based	[22](*), [42], [39], [35]	[16], [38](*), [24], [19]
without agents	[3], [2]	[17], [45], [25], [34], [31] [46], [43], [44](*), [33]

The above table gives another type of classification according to the criterion whether an approach uses agents or not ([11] identifies essentials of different multi-agent architectural styles (MAS) and shows how MAS can be characterized and evaluated), as well as whether an approach introduces a general framework (taking in the consideration both structured P2P and unstructured P2P net-

works as well as grid). Those papers that use a very specific LB algorithm are marked with "(*)". For example [34] puts the focus on DHT-based P2P systems only. First, let us discuss the articles that offer both, *agents and framework*: [22] uses a very problem-specific LB algorithm and concentrates on parallel database systems. In [42], a dynamic model of agent-based LB on grids is presented with the goal of exploring the effects of agents' strategies on the quality of LB. The results describe the dynamic behavior of LB on grids as well as modeling and predicting LB behavior. But the model used for abstraction is not very generic. It is more an agent based LB than we can say that it is a framework in our sense. [39] is not generic and in particular designed for parallel database systems. [35] uses agents and states that the approach is a generic one to implement any kind of dynamic LB algorithm in a heterogeneous cluster using software agents. However, it strictly uses only sender-initiated algorithms and no generalization is shown that allows plugging in also other algorithms.

Second, one relevant representative of each other category is presented: *Agents, no framework:* [19] presents an extension of the AMBLE model, an awareness model which manages LB by means of a multi-agent based architecture, with the aim to establish a cooperative LB model for collaborative grid environments. This model, named C-AMBLE (Cooperative Awareness Model for Balancing the Load in grid Environments) applies some theoretical principles of multi-agents systems, awareness models, and third party models, to promote an efficient autonomic cooperative task delivery in grid environments. *No agents, framework:* In [2], the design of a flexible LB framework is described and runtime software system for supporting the development of adaptive applications on distributed-memory parallel computers, i.e., the Implicit Load Balancing component of the PREMA runtime system is presented. An indication of the flexibility of the PREMA system has been given by implementing several LB policies. The four policies shown here scratch the surface of the scheduling methods possible. *No agents, no framework:* In [33], LB at the middleware level allows more flexibility than existing solutions based at lower system levels. However, it requires an execution infrastructure and mechanisms to be integrated seamlessly. DLBS (Dynamic Load Balancing Service) brings new solutions regarding large scale LB for middleware-based applications. DLBS offers a multi-criteria and easily customizable LB service. It consists of a scalable monitoring infrastructure, a connection manager (integrated into the middleware) and customizable LB strategies.

3 SILBA Pattern

The SILBA pattern is domain independent and can be used on different levels: local, with static and dynamic routing, and combining several routing strategies. It is composed of several sub-patterns that are described in this section. Its basic principle is autonomous agents that operate in a peer-to-peer network with a dynamically changing amount of work and decide on their own when to pick up or push back work. Let us first define the necessary notions, components, and assumptions used for the definition of the pattern:

A **peer node** (or node for short) is a computing device that itself might consist of several core processors.

A **request** consists of a task to be performed given in a standard format like XML or WSDL, a request identifier, a client identifier, a role describing the capacities and skills a worker must have in order to be able to process the task, a priority given in absolute terms, an informal or semantic description of the task, a timeout date, and a URL indicating a location where the answer shall be put.

An **autonomous agent** (or agent for short) is a software program that is self-responsible to be up and running. An agent implements a certain reactive and continuous behaviour [12]. Agents can move from node to node, and they can dynamically join and leave. So-called *worker agents* perform requests, decide autonomously when they act and which request they take (first or at all). Different roles determine which requests they may execute. At a certain point in time, a worker agent is associated with a node. If it fails, an automatic fail-over shall take place. Other agents occurring in SILBA are *allocation agents* and *routing agents* (see section 3.2 and 3.3).

A **client** issues requests at any reachable node in the network with an indicator, how it wants to be informed about the results and where the results shall be placed. The load that is produced by the clients can vary and peak times must be handled through the addition of resources (e.g. agents and computing nodes). Clients may either synchronously wait for the answer, or continue their job, picking up the answer later (asynchronicity). Thus, a client might go off-line and get the answer later, if they are re-connected.

In this paper, we define a **network** to connect nodes logically thus forming an overlay. Networks can be clustered into "super networks". However, as we underneath have to assume physical network links, a network can be disrupted and become partitioned into sub-networks.

A **design pattern** [21] describes a recurring and reusable solution in software design. We are mainly interested in inter-enterprise patterns for LB.

By **framework** we understand the implementation of a complex pattern – typically composed of several sub-patterns – that represents a general, re-usable architectural solution to a certain problem scenario, in our case LB. A framework can be measured by its architecture agility.

Blackboard based communication offers a high degree of decoupling (time, reference, space) and supports agents' autonomy. Clients and agents need not know each other and can act autonomously. We therefore will follow a space-based architectural style for the design of the single SILBA patterns where the entire communication is carried out via shared spaces. The composition of space-based patterns will be done by using standardized entry formats and interaction patterns in shared spaces.

In the following we describe several patterns that finally can be composed to a LB framework.

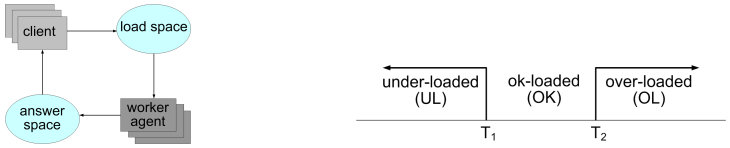

Fig. 1. (a) Local node pattern. (b) Node classification according to transfer policy.

3.1 Local Node Pattern

The local node pattern is responsible to model load generation at one node and the execution of requests by local worker agents. Workers actively compete for work. If a worker agent fails the work shall be taken over by another one. The basic components of the local node pattern (see fig. 1 (a)) are: clients, worker agents, load space, and answer space.

A **load space** is a place where new requests are put by clients, and where the information about all worker agents' registrations and the current load status of the node are maintained. A load originates from client(s).The requests shall be accessible in either the order they arrived, or by means of other criteria like e.g. their priority, the required worker role, or their timeout date. The load space can be queried about its current load status which can be either of under-loaded (UL), ok-loaded (OK), or over-loaded (OL) (see fig. 1 (b)). The determination about the load status is done according to a certain policy (see section 3.2) called transfer policy that shall be commutable.

An **answer space** is a place where the answers that were computed by the worker agents are put and where they can be picked up by the clients. The computation result is sent directly (not routed) to the answer space.

3.2 Allocation Pattern

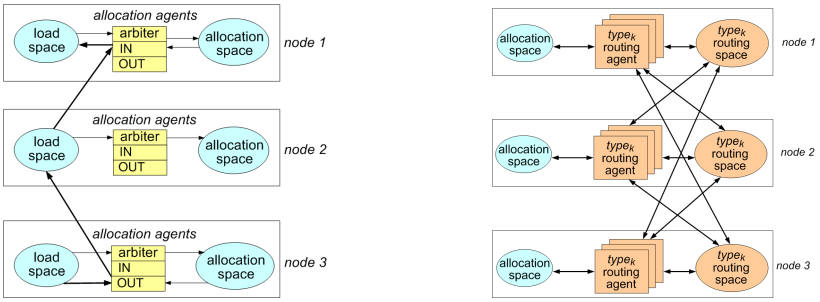

Fig. 2. (a) Allocation pattern. (b) Routing pattern.

The arbiter pattern is responsible to decide about LB and to redirect load between the load spaces of different local nodes. The basic components of the

arbiter pattern (see fig. 2 (a)) 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.

Transfer policy determines whether and in which form a resource should participate in load distribution and in that sense, the classification of resources is done [37]. A simple transfer policy would be to define two parameter values T_1 and T_2 (see fig. 1 (b)) which can either be assumed to be static or can be changed dynamically.

Location policy determines a suitable partner of a particular resource for LB [37]. 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 so-called routing agents (see section 3.3). For the latter case, the allocation space can also be used to store information about decisions and subsequent resulting routing demands of the arbiter agent, i.e. whether load shall be get rid off or new load shall be fetched from other nodes. This might cause routing agents to become active.

3.3 Routing Pattern

The routing pattern is responsible to execute the location policy according to a special LB algorithm. The basic components of the arbiter pattern (see fig. 2 (b)) are: allocation space, routing agents, and routing space.

Routing agents perform a special routing strategy to implement a certain LB algorithm. They are either always active and continuously do their jobs, or triggered upon requests of the arbiter allocation agent. In both cases they perform the location policy by communicating with other routing agents of the same type, which thus form a dynamically structured overlay network [1].

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. Here, specific information as required by the applied algorithm is stored and retrieved, e.g. pheromones if we use ants, or duration of waggle dance if we use bees [36]. Eventually, the information about a best or suitable partner node is stored in an allocation space where the corresponding IN or OUT allocation agents can grab this information and distribute the load between the local node and its partner node.

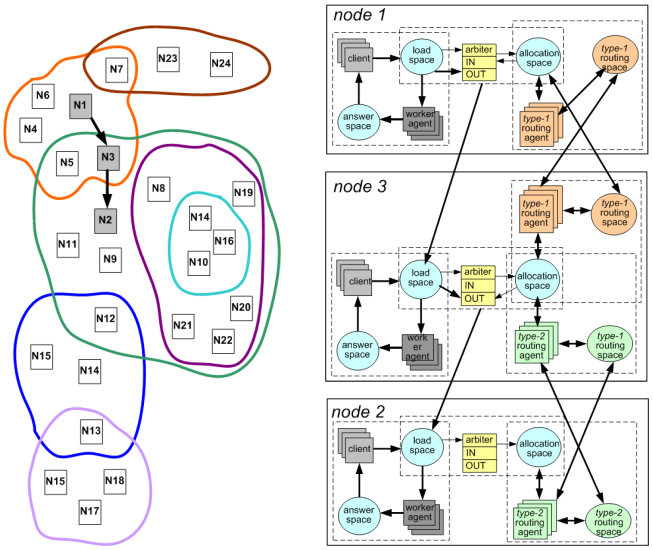

Fig. 3. (a) Network topology configuration example. (b) Pattern composition.

3.4 Pattern Composition

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. E.g. to combine the local node pattern with the allocation pattern both must share the same load space. The arbiter agent must be able to query the load information and the IN and OUT allocation agents must understand the request entries written by the clients. This kind of composition offers more flexibility than the composition possibilities in message exchange patterns as supported by enterprise service bus middleware, where basically assemblies of patterns are build. With SILBA patterns, bi-directional control flows are possible and arbitrary logical network configurations can be easily constructed (dynamically). Fig. 3 (a) depicts 7 networks that have different relationships to each other. Nodes can belong to one or more networks, e.g., nodes N1 and N2 are part of one network each, whereas N3 belongs to two networks. Fig. 3 (b) shows the composition of the mentioned nodes N1, N2, and N3 in more detail.

4 Implementation with a Space-Based Middleware

We use a space-based architecture (extensible virtual shared memory[1] [10]) that generalizes Linda tuple based communication [15] as well as several concepts that have been proposed in the literature to make the space extensible (e.g. reactions

[1] The Java based open source implementation is available at www.xvsm.org.

[32], programmable behavior ([6], [13]), introduction of priority and probability tags [4]) in one single concept called *shared containers* [28].

4.1 Shared Containers and Scheduler

A shared container is a not-nested sub-space that maintains *entries*. An entry is formally defined as a multiset of labeled values called properties [10]. A property can be used to represent either a payload or meta-data relevant for coordination. This way the entry offers a clear separation between user data (cf. message body) and coordination data (cf. message head). Meta-model operations are: create-container, destroy-container; access operations are like in the Linda model [15] (where they are termed rd, out, and in): read, write, take, and destroy; transactional operations are: transaction-start, transaction-commit, and transaction-rollback. Operations may take a timeout parameter, specifying the time after which the operation is de-scheduled from the core with an exception, if it was not fulfilled until then. Timeout of transaction-start is an important feature that issues the entire transaction to abort if the timeout has expired. Each container has one or more *coordinators* that define the exact semantics of each operation. A coordinator is a user defineable coordination mechanism that specifies how the data in the container are accessed. Typical examples include fifo order, key selection, access via label names using the built-in relational query operators [10], template matching (cf. Linda), RDF queries, or XML querying facilities [40].

A shared data space is a collection of containers. A container is addressed via its URL in the internet. A lookup mechanism resolves a published container name to its URL. A container can refer to other (sub)containers forming a more complex coordination data structure. The asynchronous and blackboard based space-based communication model allows programmers to explicitly control interactions among processes via shared data. It is advantageous to serve for the collaboration of autonomic (multi) agents as it avoids a coupling through direct interactions between the agents ([6], [32]). All interactions of an agent are carried out via its environment [23] in a symmetric and autonomous way.

As a further extension to the classical Linda model, a scheduling mechanism exists that can start agents at a certain, dynamically configurable time schedule. The migration of an agent can therefore easily be mapped into requesting a new agent that has access to the same shared containers at another site, and then terminating the original one.

Each SILBA space is realized by an addressable container. For the implementation we assume that every container can be reached in the Internet. All containers provide the navigational, built-in XVSM query language [10] (relational operators, sorting, and counting).

4.2 Local Node Implementation

Load space is a container with an implicit coordinator termed *load coordinator*, and the answer space uses a key coordinator for the requestID. The load coor-

dinator keeps track of every request that is inserted, when it is removed, and of all worker registrations. It implements a policy that decides about UL, OK, or OL of this node depending on the current number of workers and requests. Of course, more sophisticated transfer policies can be applied, that also take into consideration statistics information as well as operating system load, and hardware specifications. As the coordinator is pluggable, this policy can be changed at any time, even dynamically.

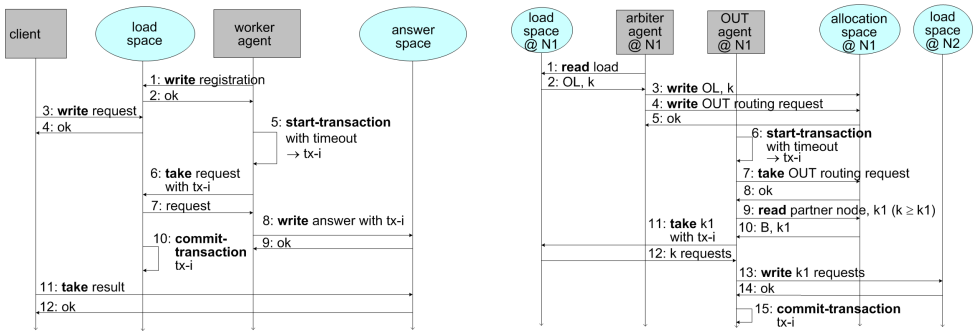

Fig. 4. (a) Local node agent interaction. (b) Allocation agent interaction.

Fig. 4 (a) shows a scenario of agent collaboration at local node level. First, each worker agent must register at the load space. It will e.g. write the following entry to the load space[2]: "[**workerId**:myID, **workerRole**:translator]". Then clients can issue requests by writing tuple like "[**job**:compile, **params**:myProg, **description**:"compile a java program", **timeout**:100, **answerURL**:myURL, **reqID**:123, **clientID**:456,**workerRole**:JavaCompiler]" into the load space.

The worker agents compete for tasks but only one will be able to execute the take operation using a new local transaction tx-i, execute the task, write the result as answer entry of the form "[**result**:resultData, **statusCode**:ok, **reqID**:123, **clientID**:456]" into the answer space using tx-i, and finally commit tx-i. If a worker that fails after having called take request and before committing the transaction tx-i, the timeout given at transaction start will fire and cause the rollback of tx-i. Failover is achieved in that another worker can take the request. Finally, the client takes the result from the answer container, correlating it with its request via the request ID, using the key coordinator for that.

4.3 Allocation Implementation

Fig. 4 (b) shows an interaction scenario of an OUT allocation agent. The arbiter agent continuously reads the load information from the load space, using the

[2] Property labels that have been standardized for the LB scenario and must be know by agents for the interoperability of the patterns are written in boldface.

load coordinator of the container. It writes this information into the routing space. If the result is OL, it also generates an OUT routing request. The OUT agent watches for outgoing routing requests, in a newly created transaction takes a next routing request, tries to read a partner information from the allocation space, and if found, takes k1 (k1 $\leq$ k) requests from its load space and transfers them to the found partner site, and finally commits its local transaction. If no outgoing routing request is found, or if not (yet) partner information exists, it will abort the transaction and try later. For the case that the worker crashes, a transaction timeout is used at transaction start.

4.4 Routing Implementation

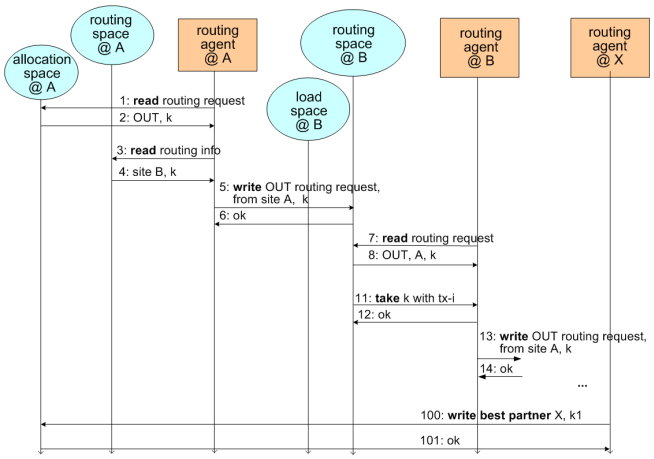

Fig. 5. Routing agent interaction.

Fig. 5 shows a basic routing scheme started by node A in that an OUT routing request is found in the allocation space. The routing agent at site A – performing a certain strategy - reads this request, then it reads the routing information from its dedicated routing space (it will find here one or more neighbors depending on which algorithm it implements, e.g. ants [37], or bees ([8], [29], [36])), and routes the request to the neighbor(s). The routing agent at site B behaves in the same way and this goes so on until a routing agent at a site X finds out that its local node is OK or UL and can accept a certain amount k1 (k1 $\leq$ k) of requests. It will write this information directly back in a P2P manner to the originally requesting site A. This way, the location policy that resolves requests for partner nodes is implemented.

5 Examples and Benchmarks

5.1 Mapping Some Algorithms to SILBA

To demonstrate the agility of the SILBA pattern based framework, we show that algorithms can be easily exchanged, and mapped some well-known LB algorithms to SILBA:

1) *Random/Round Robin Algorithm.* Some neighbors of the current node are statically stored in the routing space at initialization phase. These neighbors are retrieved from the container, one of them is chosen randomly and the task is scheduled. Similarly, Round Robin can be mapped.

2) *Sender/Receiver/Symmetric Algorithm.* For the Sender algorithm, the OL node (Sender) initiates the routing. This can be achieved by configuring just an OUT allocation agent, but no IN allocation agent. Analogously, for Receiver algorithm, where the UL starts to look for available tasks, an IN agent must be scheduled at the corresponding local nodes XVSM runtime. The Symmetric algorithm can be mapped by combining Sender and Receiver configurations.

7) *Adapted Genetic Algorithm (GA).*

We adapted the algorithm suggested in [47], i.e. we have no central coordinator. The routing agents of nodes implement the algorithm that means we have several GAs, performed on different nodes concurrently running. The description of the sliding window technique is remodulated – we can say that the size of window(s) is the same for all GAs and fixed to the number of nodes, where every node has the current (waiting) task as its candidate in that window. Every GA will obtain some combination according to the fitness function – they can be compared and the best one can be chosen as the final combination (at time t). The issue who will be appointed to do the actions of "comparing" and "choosing" is also done by the routing agents. They take care of that, communicate and exchange this information. Therefore, GA fires continously, i.e., every OL node and UL node can trigger GA and the procedure is repeated until all requests are done. In our case, the sliding window should not be directly dependable on a client who can put tasks somewhere in the network. A node has its load space for waiting (and new-coming) tasks which will be filled with requests from its node plus from all nodes of reachable networks each time when GA is triggered.

5.2 Benchmarks

Algorithm	# Nodes	in ms
Round Robin	16	19893160
Sender	16	18721100
Adapted Genetic Alg.	16	9397000

Fig. 6. Benchmarking three different LB Algorithms on 16 Nodes.

Node1	Node2	Node3	Node4	Node5	Node6	Node7	Node8	Node9	Node10	Node11	Node12	Node13	Node14	Node15	Node16
385100	248340	268530	573520	614370	375390	528520	296840	461530	349530	495260	385170	253960	29576790	368160	445830
7567000	9125000	4231000	9962000	12843000	2045000	8947000	3241000	9780000	12125000	11286000	9300000	9015000	450724000	8043000	9662000
55876000	4704000	4698000	4255000	8145000	4053000	2968000	3716000	4067000	4164000	4301000	4272000	4882000	4578000	3748000	3109000

Fig. 7. Row 1: Round Robin. Row 2: Sender. Row 3: Adapted GA.

As a proof of concept that the SILBA implementation produces reasonable results, in a first step these three different algorithms (Round Robin, Sender Negotiation, and Adapted GA) for location policies have also been implemented and benchmarked. The transfer policy (implemented by the load coordinator) was realized by relatively determined values T_1 and T_2.

The benchmarks were performed on a cluster of 4 machines. Each machine had the following characteristics: 2*Quad AMD 2,0GHz with 16 GB RAM. We simulated the network of 16 (virtual) nodes. At the beginning of each test runs, the starting state was "cold start" and all nodes were UL. The maximum number of tasks (that represented the load in the system) was 400. The obtained results can be seen in figs. 6 and 7. The benchmark time is the makespan.

Fig. 7 shows the distribution of tasks time per node, i.e. how much each node was active during the execution of a particular algorithm and how much load it had. Note: many tasks have been done in parallel, so that's why the complete benchmark(s) time(s) is (are) less than some times in fig. 7. This figure provide an additional information and analisys about the "behavior" of different algorithms, but the real comparison of algorithms is shown in fig. 6.

As a result, Round Robin algorithm activates all nodes in the network, whereas using Sender algorithm all work was done by activating only a few nodes and the rest of the network was idle. During the execution of the location policy by using the Round Robin algorithm, all nodes (except node 14) were in the state OK, and node 14 changed its state from OK to OL and vice versa. During the execution of location policy by using Sender algorithm, the majority of nodes were not activated, node 8 participated and stayed in OK state all the time, where nodes 1 and 14 changed their state from OK to OL and vice versa.

6 Conclusion

We presented a generic and composable load balancing (LB) pattern for worker agents having access to either one local node, or many nodes in the intranet and in the internet. The intention was to develop a self-organized LB architecture, extensible towards different LB strategies and adaptable to any kind of domain-specific problems. This objective could be solved by basing the pattern rather on a space-based architecture style than on classical message exchange patterns. The space-based collaboration allows for a high decoupling of the agents and enables their autonomic behavior. Transfer and location policies can be plugged in by means of coordinators and routing agents. Worker agents are active and autonomously deciding whether and which task to take, and whether to execute

it or not; routing agents exhibit an autonomic behavior, if the transfer and location policies are based on intelligent algorithms that support self-organization. This general pattern can therefore easily be populated with a wide range of LB algorithms. New resources can be dynamically added or removed to the resource-pool at any time and the failure of one worker agent or node does not jeopardize the functioning of the whole system.

We have carried out first benchmarks that proved that an approach using active agents can be adapted to any underlying algorithm. This justifies the design of SILBA as a pattern that supports autonomic agents. The pattern supports nested networks at any level through composition. Even a mixed usage of strategies is possible. Moreover, further tuning parameters can easily be added, as the data entries and structures in the coordination space are also extensible. We considered only the time in our benchmarks, and intend to treat other important aspects of load balancing (cost of communication, communication delay etc.) in future investigations, to work on different types of metrics, to implement intelligent algorithms (e.g., swarm intelligence as proposed in [36]), and to benchmark and compare different instantiations of SILBA with each other as well as with other existing algorithms for load balancing.

Acknowledgements. We would like to thank Fabian Fischer for implementing the SILBA pattern and performing the benchmarks, and Richard Mordinyi for his helpful comments on this text.

References

1. S. Androutsellis-Theotokis and D. Spinellis. A survey of peer-to-peer content distribution technologies. *ACM Comput. Surv.*, 36(4):335–371, 2004.
2. K. Barker, A. Chernikov, N. Chrisochoides, and K. Pingali. A load balancing framework for adaptive and asynchronous applications. *IEEE Transactions on Parallel and Distributed Systems*, 15(2):183–192, 2004.
3. K. J. Barker and N. P. Chrisochoides. An evaluation of a framework for the dynamic load balancing of highly adaptive and irregular parallel applications. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 45, 2003.
4. M. Bravetti, R. Gorrieri, R. Lucchi, and G. Zavattaro. Quantitative information in the tuple space coordination model. *Theor. Comput. Sci.*, 346(1):28–57, 2005.
5. A. G. Bronevich and W. Meyer. Load balancing algorithms based on gradient methods and their analysis through algebraic graph theory. *J. Parallel Distrib. Comput.*, 68(2):209–220, 2008.
6. G. Cabri, L. Leonardi, and F. Zambonelli. Mars: A programmable coordination architecture for mobile agents. *IEEE Internet Computing*, 4(4):26–35, 2000.
7. J.-C. Chen, G.-X. Liao, S. Hsie, and C.-H. Liao. A study of the contribution made by evolutionary learning on dynamic load-balancing problems in distributed computing systems. *Expert Syst. Appl.*, 34(1):357–365, 2008.
8. C. S. Chong, A. I. Sivakumar, M. Y. H. Low, and K. L. Gay. A bee colony optimization algorithm to job shop scheduling. In *WSC '06: Proc. of the 38th conference on Winter simulation*, pages 1954–1961. Winter Simulation Conf., 2006.
9. A. Cortés, A. Ripoll, F. Cedó, M. A. Senar, and E. Luque. An asynchronous and iterative load balancing algorithm for discrete load model. *J. Parallel Distrib. Comput.*, 62(12):1729–1746, 2002.

10. S. Craß, E. Kühn, and G. Salzer. Algebraic foundation of a data model for an extensible space-based collaboration protocol. In *13th International Database Engineering & Applications Symposium (IDEAS)*, 2009, to appear.
11. P. Davidsson, S. Johansson, and M. Svahnberg. Characterization and evaluation of multi-agent systems architectural styles. *Software Engineering for Multi-Agent Systems IV*, 3914:179–188, 2006.
12. S. Dobson, S. Denazis, A. Fernández, D. Gaïti, E. Gelenbe, F. Massacci, P. Nixon, F. Saffre, N. Schmidt, and F. Zambonelli. A survey of autonomic communications. *ACM Trans. Auton. Adapt. Syst.*, 1(2):223–259, 2006.
13. S. Ducasse, T. Hofmann, and O. Nierstrasz. Openspaces: An object-oriented framework for reconfigurable coordination spaces. In *Coordination Languages and Models*, volume 1906, pages 1–19, 2000.
14. D. L. Eager, E. D. Lazowska, and J. Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Trans. Softw. Eng.*, 12(5):662–675, 1986.
15. D. Gelernter and N. Carriero. Coordination languages and their significance. *Commun. ACM*, 35(2):97–107, 1992.
16. C. Georgousopoulos and O. F. Rana. Combining state and model-based approaches for mobile agent load balancing. In *SAC '03: Proceedings of the 2003 ACM symposium on Applied computing*, pages 878–885, 2003.
17. B. Godfrey, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica. Load balancing in dynamic structured P2P systems. In *Proc. IEEE INFOCOM*, 2004.
18. J. Gomoluch and M. Schroeder. Information agents on the move: A survey with load balancing with mobile agents. *Software Focus*, 2(2), 2001.
19. P. Herrero, J. L. Bosque, and M. S. Pérez. An agents-based cooperative awareness model to cover load balancing delivery in grid environments. In *OTM Workshops (1)*, pages 64–74, 2007.
20. C.K. Ho and H.T. Ewe. Ant colony optimization approaches for the dynamic load-balanced clustering problem in ad hoc networks. In *Swarm Intelligence Symposium, Hawaii*, pages 76–83. IEEE, 2007.
21. G. Hohpe and B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, 2003.
22. T.-L. Hu, G. Chen, K. Chen, and J.-X. Dong. An adaptive load balancing framework for parallel database systems based on collaborative agents. volume 1, pages 464–468. IEEE, 2005.
23. N. Janssens, E. Steegmans, T. Holvoet, and P. Verbaeten. An agent design method promoting separation between computation and coordination. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 456–461, 2004.
24. S. Johansson, P. Davidsson, and M. Kristell. Four multi-agent architectures for intelligent network load management. In *4th Int. Workshop on Mobile Agents for Telecommunication Applications (MATA)*, pages 239–248, London, UK, 2002. Springer-Verlag.
25. D. R. Karger and M. Ruhl. Simple efficient load balancing algorithms for peer-to-peer systems. In *SPAA '04: Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 36–43, 2004.
26. A. Keren and A. Barak. Adaptive placement of parallel java agents in a scalable computing cluster. volume 10, pages 971–976, 1998.
27. P. Krueger and N. G. Shivaratri. Adaptive location policies for global scheduling. *IEEE Trans. Softw. Eng.*, 20(6):432–444, 1994.
28. E. Kühn, R. Mordinyi, L. Keszthelyi, and C. Schreiber. Introducing the concept of customizable structured spaces for agent coordination in the production automa-

tion domain. In *the 8th Int.Conference on Autonomous Agents and Multiagent Systems (AAMAS 2009)*, 2008.
29. N. Lemmens, S. de Jong, K. Tuyls, and A. Nowé. Bee behaviour in multi-agent systems. pages 145–156. 2008.
30. F. C. H. Lin and R. M. Keller. The gradient model load balancing method. *IEEE Trans. Softw. Eng.*, 13(1):32–38, 1987.
31. Y. Murata, H. Takizawa, T. Inaba, and H. Kobayashi. A distributed and cooperative load balancing mechanism for large-scale p2p systems. In *SAINT-W '06: Proceedings of the International Symposium on Applications on Internet Workshops*, pages 126–129. IEEE, 2006.
32. G. Pietro Picco, A. L. Murphy, and G.-C. Roman. Lime: Linda meets mobility. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 368–377. IEEE, 1999.
33. E. Putrycz. Design and implementation of a portable and adaptable load balancing framework. In *CASCON '03: Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative research*, pages 238–252. IBM Press, 2003.
34. M. A. Rahman. Load balancing in dht based p2p networks. In *5th Int. Conference on Electrical and Computer Engineering, ICECE 2008*, pages 164–171, 2008.
35. A. Rajagopalan and S. Hariri. An agent based dynamic load balancing system. pages 164–171, 2000.
36. V. Sesum-Cavic and E. Kühna. Instantiation of a generic model for load balancing with intelligent algorithms. In *IWSOS '08: Proceedings of the 3rd International Workshop on Self-Organizing Systems*, pages 311–317. Springer, 2008.
37. T. Stützle and H. Hoos. Max-min ant system. *Future Generation Comput. Syst.*, 16(9):889–914, 2000.
38. H. Thant, K. San, K. Tun, T. Naing, and N. Thein. Mobile agents based load balancing method for parallel applications. In *6th Asia -Pacific Symposium on Information and Telecommunication Technologies (APSITT 2005)*, 2005.
39. J. Tian, Y. Liu, X.-H. Yang, and R. Du. Design and analysis of a novel load-balancing model based on mobile agent. In *Advances in Machine Learning and Cybernetics, 4th International Conference (ICMLC)*, pages 70–80, 2005.
40. R. Tolksdorf and R. Menezes. Using swarm intelligence in linda systems. In *Int. Workshop Engineering Societies in the Agents' World (ESAW)*, pages 49–65, 2003.
41. H. Une and F. Qian. Network load balancing algorithm using ants computing. In *IAT '03: Proceedings of the IEEE/WIC International Conference on Intelligent Agent Technology*, page 428, 2003.
42. Y. Wang and J. Liu. Macroscopic model of agent-based load balancing on grids. In *AAMAS '03: Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, pages 804–811. ACM, 2003.
43. M. Xu and J. Guan. Routing based load balancing for unstructured p2p networks. *Future Generation Communication and Networking*, 2:332–337, 2007.
44. Z. Xu and L. Bhuyan. Effective load balancing in p2p systems. In *CCGRID '06: Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid*, pages 81–88. IEEE, 2006.
45. Y. Zhu and Y. Hu. Efficient, proximity-aware load balancing for dht-based p2p systems. *IEEE Trans. on Parallel and Distributed Systems*, 16(4):349–361, 2005.
46. S. Zoels, Z. Despotovic, and W. Kellerer. Load balancing in a hierarchical dht-based p2p system. pages 353–361. IEEE, 2007.
47. A. Y. Zomaya and Y.-H. Teh. Observations on using genetic algorithms for dynamic load-balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(9):899–911, 2001.