Computing and Informatics, Vol. 25, 2006, 17–42

COMPARING A TRADITIONAL AND A MULTI-AGENT LOAD-BALANCING SYSTEM

Andraz Bezek, Matjaz Gams

Department of Intelligent Systems
Jozef Stefan Institute
Jamova 39, 1000, Ljubljana, Slovenia
e-mail: {andraz.bezek, matjaz.gams}@ijs.si

Manuscript received 14 January 2005; revised 9 January 2006
Communicated by Ladislav Hluchý

Abstract. This article presents a comparison between agent and non-agent based approaches to building network-load-balancing systems. In particular, two large software systems are compared, one traditional and the other agent-based, both performing the same load balancing functions. Due to the two different architectures, several differences emerge. The differences are analyzed theoretically and practically in terms of design, scalability and fault-tolerance. The advantages and disadvantages of both approaches are presented by combining an analysis of the system and gathering the experience of designers, developers and users. Traditionally, designers specify rigid software structure, while for multi-agent systems the emphasis is on specifying the different tasks and roles, as well as the interconnections between the agents that cooperate autonomously and simultaneously. The major advantages of the multi-agent approach are the introduced abstract design layers and, as a consequence, the more comprehendible top-level design, the increased redundancy, and the improved fault tolerance. The major improvement in performance due to the agent architecture is observed in the case of one or more failed computers. Although the agent-oriented design might not be a silver bullet for building large distributed systems, our analysis and application confirm that it does have a number of advantages over non-agent approaches.

Keywords: Multi-agent systems, distributed systems, load-balancing

1 INTRODUCTION

Building reliable, large internet services is a difficult task, made more so by the requirements for a virtually continuous uptime together with a consistent response time [20]. Stable services must be able to cope with many undesirable factors such as the explosive growth of traffic over the internet [4] and possible hardware and software failures. The widely accepted solution for the ever-increasing traffic is a network-load-balancing system, which balances the incoming network traffic among a cluster of servers. It is a requirement that such systems are fault-tolerant as well as scalable, i.e., that they can be resized and reconfigured. Agent-oriented software engineering (AOSE) [26, 27] has received a lot of attention as a potential mainstream initiative for distributed software engineering [11, 12, 13, 16, 17], however it has not been fully introduced as a major commercial software engineering paradigm.

A multi-agent system (MAS) is a loosely coupled network of software entities that work together to solve problems that are beyond the individual capabilities or knowledge of each entity [5]. A MAS is a distributed reactive system [21] that maintains an ongoing interaction with the environment.

The structure of this paper is as follows: A traditional and a multi-agent application of a fault-tolerant network-load-balancing system are described in Section 2, together with the design and implementation aspects. In Section 3, the fault-tolerant characteristics of load-balancing systems and the differences between traditional and agent load-balancing systems are presented. In Section 4, detailed analyses, important observations, and a comparison with the non-agent version are described. Related work is presented in Section 5. Finally, the conclusions are presented in Section 6.

2 THE TRADITIONAL AND AGENT-BASED LOAD-BALANCING SYSTEMS

The term load balancing (or LB for short) is used to refer to network load balancing or – more precisely – load balancing of IP packets, also known as IP-level load balancing.

The term cluster means all the computers within a LB system, i.e., all the servers, the load balancers and the administration computers. Sometimes we refer to a cluster of servers and a cluster of load balancers, each describing a set of computers within the cluster with server/LB functionality.

The purpose of a network-LB system is to evenly distribute the incoming network traffic among servers in a cluster. The distribution is done according to the desired LB policy, which often takes into account performance metrics such as the amount of network traffic or the processor load.

In this section we present a traditional LB system and an agent-based LB system for balancing IP traffic between servers. The traditional LB system is described in Section 2.1. In Section 2.2 we describe the agent-based LB system together with agent-oriented analysis and design. We do this by adopting the basic concepts of AOSE. Both systems are fully operational and applied.

2.1 Traditional Commercial Load-Balancing System

The IP-level LB system (LB at network layer 3 in the OSI Model), designed in a traditional way, consists of a single software program running on all the machines in the cluster. Its operation mode as a load balancer or server-side software is defined in a configuration file, which must reside on each computer. All configuration changes except server addition result in a cluster restart. This time-consuming operation forces all programs to restart and re-read the configuration. It is the user's responsibility to distribute and synchronize the file configuration within the cluster. Actual load balancing is performed by a loadable Linux kernel module.

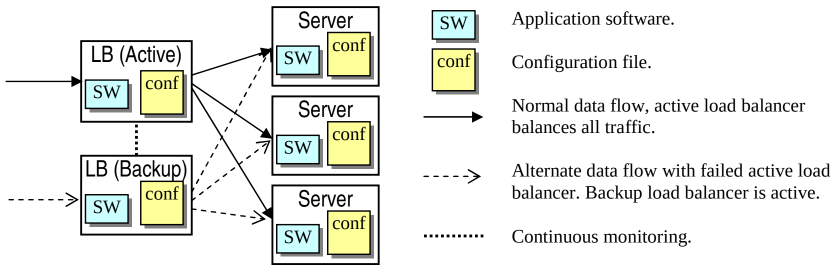

LB (Active)
SW
conf
LB (Backup)
SW
conf
Server
SW
conf
Server
SW
conf
Server
SW
conf
SW
Application software.
conf
Configuration file.
Normal data flow, active load balancer balances all traffic.
Alternate data flow with failed active load balancer. Backup load balancer is active.
Continuous monitoring.

Fig. 1. Architecture of a traditional LB system

Although the nature of IP routing requires only one computer to perform the LB, it is also possible to employ an active/backup model of redundancy, as presented in Figure 1. With it the software on the backup LB machines monitors the operation of the active LB computer, as presented in Figure 4 a). In the case of a failure, shown in Figure 4 b), the software on the backup computers votes for a new active LB computer that takes over the complete functionality of the failed one. The system also periodically checks for server and service availability through hard-coded checking routines. If a malfunction is detected, the server or service is removed from the LB process.

2.2 Agent Load-Balancing System

The AOSE approach we used is based on the MASSIVE process model presented by Lind [15]. It is a pragmatic process model for the development of multi-agent systems based on a combination of standard software-engineering techniques. The terminological framework is based on the so-called *views*. A view represents a set of conceptually linked features that can be regarded as an abstraction of the model with respect to a specific point of view. Usually, several abstractions from the same reality exist, and a collection of views achieves a logical decomposition of the target system. This section describes only two of them: *the role view* and *the interaction view*. The former determines the functional aggregation of the basic problem-solving capabilities and is sufficient to present the basic properties of the system; the latter defines interaction as the fundamental concept within a system that is comprised of multiple independent entities that must coordinate themselves in order to achieve their individual as well as joint goals.

The abstract system specification is the same for the non-agent and the multi-agent versions of a LB system. Its abstract goals follow the system specification and functional requirements, i.e., efficient LB, scalability, fault-tolerance, and manageability. Consequently, high-level tasks are somewhat similar to tasks in a non-agent system, i.e., the tasks of LB, serving and administration. We can thus define *role view* by defining specific roles. Conceptually, a computer in a cluster can take part in a server-, LB- or management-related role. By analyzing distinct computer roles, we can decompose abstract tasks into more detailed ones that can be represented as distinct agent roles.

Since the primary purpose of a server is to run services, agent roles can be defined as *server-centric*, i.e., performing computer administration and reporting of server-related statistics, and as *service-centric*, i.e., performing service management, service startup/shutdown, reporting of service-related statistics, and synchronization of service-related data. The roles hosted on LB computers are *LB-centric*, which involves control of the LB software and enforcement of the desired LB policy, and *cluster-centric*, which includes cluster-wide control by checking services, servers, and other load balancers for their correct operation. The computers with management status handle *configuration-related* issues such as handling the global configuration and providing some kind of user interface. In addition, agents on management computers perform an *agent supervision role*, which includes support for installation, health checking, and the termination of agents, and a *reporting role*, which includes reporting and gathering errors and other messages.

Altogether, 12 classes of agents (all with different roles) were designed. Table 1 presents all of them, together with their names, optional acronyms (in parentheses) and locations, on the first line, with a short description below. The design anticipated automatic installation of agents according to the current role of the computer (i.e. they are static in terms of computer role). However, computer roles can be dynamically changed, and the agents are thus correspondingly installed/uninstalled on each computer in the cluster by the Computer Management Agent.

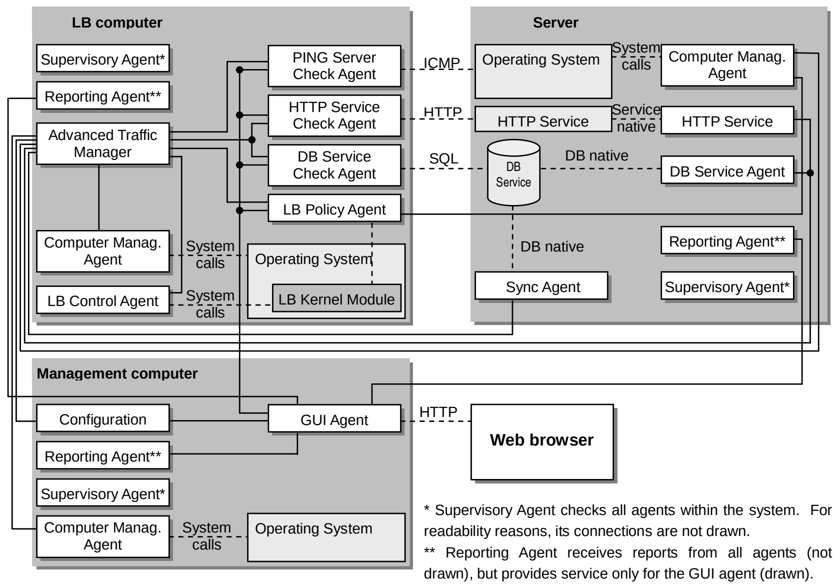
Having defined all the agent classes we need to define the agent instances as well. Each agent class can have several agent instances, while we defined separate agent instances for Service Agent, Service Check Agent, Server Check Agent, and LB Policy Agent classes. The MASSIVE model handles interaction issues through *interaction view*, where several generic forms of interaction suitable for a wide variety of contexts are defined. We analyzed all types of agent interactions: *agent-to-agent*, *agent-to-client*, and *agent-to-environment*. Designers must consider all the layers of design and construct a coherent and global agent architecture. An example design of a MAS architecture with inter-agent connections and agent locations is presented

Class	Location	Description
Advanced Traffic Manager (ATM)	LB based	A central cluster-management agent. It handles the whole cluster organization and actively checks other ATMs for proper operation. It controls the LB software according to the configuration, assigns servers and services to LB software, and periodically checks their operation. In the case of a detected failure, the faulty server or service is immediately marked as inactive.
Service Management Agent (SMA)	server based	A service-specific agent that performs all service-dependent administration. Its primary task is to manage, start, and stop specific services. It can also retrieve service-specific metrics (e.g., statistics of http accesses).
Synchronization Agent	server based	Takes care of the synchronization of service-based data. Its implementation is service specific.
LB Policy Agent	LB based	Assigns weights in LB software according to the desired LB policy. It must monitor server-based metrics (such as CPU load and the number of network connections on servers) and compute weights according to the implemented LB policy to appropriately distribute network traffic.
LB Control Agent (LBA)	LB based	Acts as a translator to LB software by translating agent commands to the specific commands of LB software and reporting status changes of the LB software.
Computer Management Agent	cluster based	Controls various computer-related tasks. It can setup network interfaces and report various metrics, such as CPU load, number of network connections, memory consumption and the amount of free disk space.
Server Check Agent	LB based	Periodically checks if a server is working properly. It is possible to implement several different server checks.
Service Check Agent	LB based	Periodically checks if a service is working properly. Each service demands a different instance of service check agent.
Configuration Agent	management based	Maintains the configuration and controls simultaneous access to it. It also broadcasts all recent configuration changes to the agent system.
GUI Agent	management based	Acts as a http server for users and an intermediate agent to the agent system. Its task is to provide a web-based user interface for cluster management, including access to configuration, messages, errors, and up-to-date information about servers, services, and LB software.
Supervisory Agent	cluster based	Starts, checks and terminates agents within one computer. According to the role defined in the configuration it must start or stop the agent operation of each computer in a cluster. In order to deal with unexpected software errors, a special agent was assigned to periodically check the health of running agents. In the case of no response, the failed agent is forcefully terminated and restarted.
Reporting Agent (RA)	cluster based	Reports various messages and errors to the user (via the GUI agent).

Table 1. Descriptions of agents in a multi-agent load-balancing system

in Figure 2, where solid lines represent communication between agents, and dashed lines represent service-specific communication with the protocol names beside.

Since the fault-tolerant computing paradigm expects failures as a rule and not as an exception, the system must be able to cope reasonably well with agent failures. In order to systematize fault-related activities we introduced different importance levels together with fault-tolerant design principles. Consequently, our design considers the following four levels of agent importance:



* Supervisory Agent checks all agents within the system. For readability reasons, its connections are not drawn.
** Reporting Agent receives reports from all agents (not drawn), but provides service only for the GUI agent (drawn).

Fig. 2. A schematic diagram of the major agent connections in a multi-agent LB system

Core agents perform tasks essential to the proper functioning of the system, which either stops or operates erroneously without them. An example of this would be the Advanced Traffic Manager, which controls vital cluster-management activities. The core agents are the ATM, the LB Control Agent, and the Configuration Agent.

Support agents carry out tasks related to the management of services, servers and agents. They are only needed for startup and shutdown activities or for the reconfiguration of a cluster. For example, the lack of the Service Management Agent would prevent the cluster from starting or stopping a certain service, but the system would remain stable. The support agents are the Computer Management Agent, the Service Management Agent, the Supervisory Agent, and the Synchronization Agent.

Regulative agents perform partial cluster optimization, and are not vital to the running of the system. For example, all checking agents are optional; the system works without them. However, such a system would be unable to detect the failures of servers and services. The regulative agents are the LB Policy Agent, the Server Check Agent, and the Service Check Agent.

User agents are needed when users require access to the cluster management. Their absence does not impact on the operation of the system. The user agents are the GUI Agent and the Reporting Agent.

Accordingly, special care was taken to develop fault-tolerant operation for core agents. The main principle used in designing these agents was to utilize redundancy. The most important agent within our system, the ATM, was designed with a special protocol to enable mutual checking of ATMs. With this protocol, backup ATMs can periodically check other active ATMs. When an active ATM fails, all backup ATMs take part in elections for a new active ATM. A newly elected ATM starts advertising the virtual IP of a running cluster. This makes it possible to introduce a virtual IP number, associated with an active LB. The traffic destined for the virtual IP number is redirected to the active ATM that hosts the operating LB software. Our design anticipates a set of virtual IP numbers for each active ATM, as presented in Figure 3 a). In the case of an ATM's failure, unused virtual IP numbers can be evenly distributed between other active LBs (illustrated in Figure 3 b)). This ensures the smooth distribution of input traffic that is taken over from a failed LB.

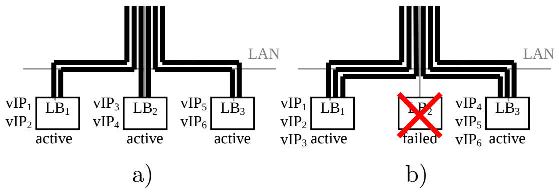


Fig. 3. Failure response of the multi-agent LB system: a) All load balancers are active at a certain time; b) In the case of failure other active load balancers share the traffic of the failed one

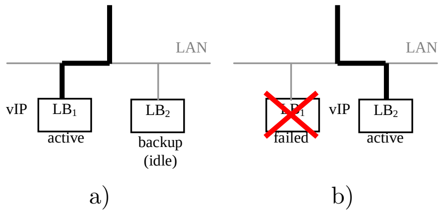
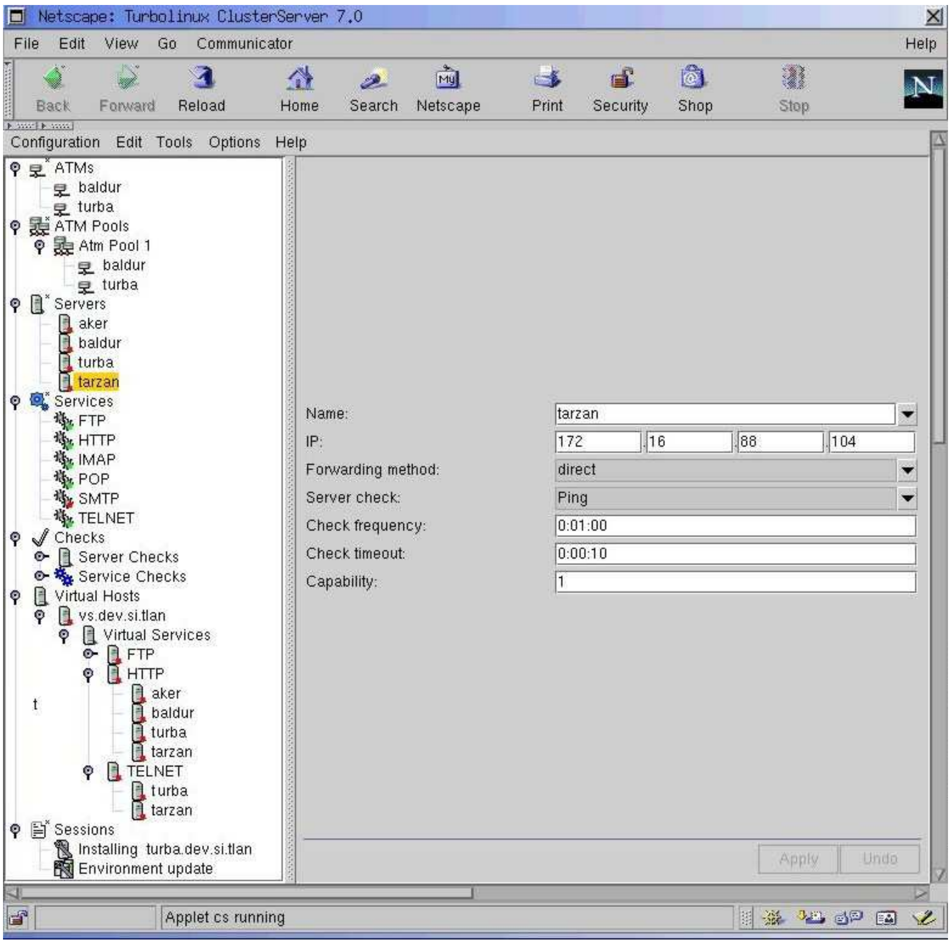


Fig. 4. Failure response of the traditional LB system: a) Only one load balancer is active at a certain time; b) Failure of the active load balancer activates the backup one

To allow redundancy, Configuration Agents must be able to operate simultaneously, thus the task of designing a Configuration Agent is similar to designing a distributed database. Having studied the problem, we concluded that simultaneous configuration commitment is unpractical and extremely rare. Consequently, the final design utilized an active/backup model of redundancy, thus allowing only one active Configuration Agent, with others running in a backup mode. All configuration changes can only be committed to the active instance. The remaining backup instances synchronize its configuration with the active one. In case that the active agent fails, the configuration can still be retrieved from the backup ones. The design of the agent system architecture is sufficient to provide all the information for the implementation stage. But before MAS implementation can take place, an appropriate agent platform must be selected. The platform defines the basic agent environment together with the agent operation and communication. The design of a particular agent platform enforces a specific multi-agent approach, which guides subsequent implementation processes. Altogether, the agent environment with its implementation and approach limits the possible implementations of a MAS, defined by the design of the agent system architecture. Although the choice of agent platform has a strong impact on the actual implementation of a MAS, its details are beyond the scope of this paper. Some agent platforms and frameworks are described in [19, 6].



Name:	tarzan
IP:	172 . 16 . 88 . 104
Forwarding method:	direct
Server check:	Ping
Check frequency:	0:01:00
Check timeout:	0:00:10
Capability:	1

Fig. 5. Web interface for managing a MAS

The separated design of agent classes and the subsequent separated implementation of agent instances introduce an important improvement over traditional programming [10]. With the latter, developers typically develop one big and complex program. Although its design can be modular and the developers develop distinct modules, it poses several implementation- and validation-related problems. During our development of a MAS we observed that developers cooperate within smaller groups. As fewer developers develop each agent, they tend to cooperate more effectively than those in bigger groups. This is a direct result of a smaller interaction overhead, which inherently adds up in the total development time. In the case of software failures, it is usually easier to test and debug smaller programs. In addition, detected software lock-ups and crashes are easier to locate in the smaller source-code footprint than those in bigger programs. Consequently, the development process is more straightforward, and thus a little faster and less prone to errors.

3 AGENT VS. NON-AGENT LOAD BALANCING

In this section we compare the agent and non-agent based system in terms of architecture and functional properties.

3.1 Architecture

The architecture of our traditional system is presented in Figure 6 b), and that of our agent system in Figure 6 d). Other architectures are presented to give a broader overview and provide future designers more options.

In a *single-server system*, shown in Figure 6 a), the server is constantly receiving input traffic. The queries presented as input traffic are processed and the results are replied to as output traffic. In the real world, the amount of output traffic is much greater than the amount of input traffic. This means that the performance of the server is limited by transportation and computation overload. The former results in saturation of the server output traffic and the latter in a high server load. To improve the performance and to increase the saturation limit, LB systems distribute requests among a cluster of servers. A *traditional load-balancing system*, presented in Figure 6 b), consists of a single load balancer and a set of servers. Accordingly, the input flow is balanced among the servers and the resulting output traffic is redirected directly to the users. Besides the obvious performance benefits, a design like this also increases the level of fault-tolerance. Furthermore, any failed server can easily be replaced by redirecting its traffic to the other active servers.

To overcome the restrictions of a single load balancer, *distributed load-balancing systems*, shown in Figure 6 c), were introduced. They require a preceding load balancer, which distributes the input traffic among the distributed LB systems. Typically, a round-robin DNS solution is used to ensure a geographical distribution of the traffic. Load balancing at this level can employ a coarser distribution policy without any impact on the underlying systems. It is clear that the level

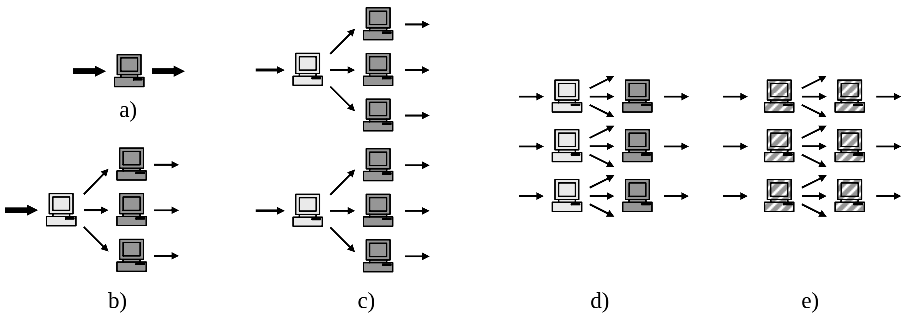


Fig. 6. Network data flow for various server systems. The light-colored computers represent the load balancers while the grey computers represent the servers: a) Single server system; b) Traditional load-balancing system; c) Distributed load-balancing system; d) Multi-agent load-balancing system; e) Adaptive multi-agent load-balancing system

of fault-tolerance is increased. Although a failed load balancer renders the whole LB sub-system unavailable, its input traffic can be balanced among other working sub-systems by updating the preceding load balancers. Unfortunately, even though distributed LB systems employ a number of distributed sub-systems, their number is still fixed, which results in the limited scalability of the system. *Multi-agent load-balancing systems*, presented in Figure 6 d), enhance scalability by employing an arbitrary number of load balancers. A preceding load balancer, as in the previous approach, is still needed, but the main benefit is the possibility to change the size of the LB cluster. Agents can be positioned at any location, thus load balancers and servers can be distributed across the internet. This design makes fast fault detection and recovery possible by enabling LB agents to constantly monitor cluster activities. The main benefit compared to the previous approach is that the servers behind the failed load balancer can still be used by other balancers; as opposed to the distributed approach where the failure of a load balancer renders all the servers behind workless. While the multi-agent approach can utilize an arbitrary number of load balancers and servers, it is still unable to dynamically and autonomously change its configuration. An *adaptive multi-agent load-balancing system*, shown in Figure 6 e), is a society of agents where an agent can play the role of a load balancer or a server. Because these roles can be dynamically changed, the system can modify itself efficiently to optimize some performance criteria. In addition, failure detection and recovery can be generalized and thus simplified, while the homogeneous and adaptive structure greatly simplifies the management of the cluster. However, an adaptive multi-agent LB system is more complex to implement and might introduce additional computation- and communication-related overheads.

It is important to note that the architectures presented in Figure 6 d) and e) can be implemented by non-agent approaches and that the agent approach *per se* does not enforce a specific LB architecture. However, here we are concerned only with architectures presented in Figure 6 d) and e) with internal agent structure.

3.2 Fault-Tolerant Characteristics

In theory, fault-tolerant systems (chapter 7 in [24]) should not have a single point of failure and should be resistant to any failure, including hardware and software failures. According to [25], fault-tolerant services must be designed to achieve high availability, safety, maintainability, and reliability. *Availability* is defined as the percentage of time that a system is operating correctly and is available to perform its functions. *Safety* refers to the situation where a system temporarily fails to operate correctly, but nothing catastrophic happens. *Maintainability* refers to how easily a failed system can be repaired. A highly maintainable system may also show a high degree of availability, especially if failures can be detected and repaired automatically. Finally, *reliability* refers to the ability of a system to run continuously and without failure.

First, we compare LB architectures in terms of reliability. If p *denotes the reliability* of a single computer over a given time, $1 - p$ is the expected single-computer error probability, M is the number of load balancers, and N is the number of servers within the system, then we can estimate the overall reliability of different LB systems. If we assume an instantaneous error detection and an appropriate response, then the reliability of each LB system, presented in Figure 6, is as follows:

1. Single server:
$$p$$
2. Traditional load-balancing system with N servers:
$$p(1-(1-p)^N)$$
3. Distributed load-balancing system with M equal-sized sub-systems:
$$1-(1-p(1-(1-p)^{N/M}))^M$$
4. Multi-agent load-balancing system:
$$(1-(1-p)^N)(1-(1-p)^M)$$
5. Adaptive multi-agent load-balancing system:
$$(1-(1-p)^N)(1-(1-p)^M)$$

The results in Figure 7 show that the traditional LB system, Figure 6 b), is substantially less reliable than the distributed and multi-agent versions for $M = 2$ and $N = 10$. Furthermore, the multi-agent LB systems perform a little better than the distributed LB systems. This means that for a small error probability, i.e., high reliability p, and large values for N and M, distributed, multi-agent and adaptive MASs become highly reliable. One of the assumptions made in the previous analysis is that a failed computer is never repaired. This is a reasonable assumption when we analyze error probability over short intervals of time, e.g., one hour. For longer time periods, however, computers would be repaired. In practice it is a common scenario that either the system is fully operational or that one computer is not working. We therefore analyze the ratio of the average performance between a system with one failed computer and a fully operational system. It is assumed that all the computers have the same failure probability and that the system performance is proportional to the number of working servers. The performance ratios for various LB architectures are as follows:

1. Single server:
$$0$$
2. Traditional LB system with N servers:
$$\frac{N-1}{N+1}$$
3. Distributed LB system with M equal-sized sub-systems:
$$\frac{M-1}{N+M}+\frac{N-1}{N+M}$$
4. Multi-agent LB system:
$$\frac{M}{N+M}+\frac{N-1}{N+M}$$
5. Adaptive multi-agent LB system:
$$\frac{M}{N+M}+\frac{N-1}{N+M}$$

The relations for various values of N are presented in Figure 8. The multi-agent LB systems, Figure 6 d) and e), perform substantially better with one computer down than the distributed version. The distributed version is still better than the traditional LB system.

All distributed systems, including MASs, are prone to failures. Agents and resources can become unavailable due to machine crashes, communication breakdowns, and numerous other hardware and software failures. Most of the work done in fault handling for MASs deals with communication failures, while the detection and recovery from faults typically rely on the traditional techniques for failure recovery. However, the traditional fault-tolerance techniques are designed for specific situations and the introduction of a MAS requires special infrastructural support, such as support for continuous and fault-tolerant agent communication.

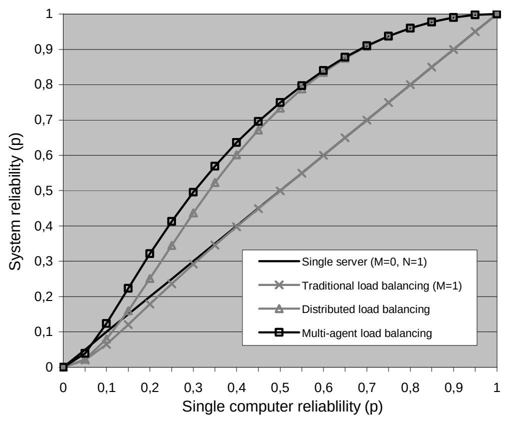


Fig. 7. System reliability for $M = 2$ and $N = 10$

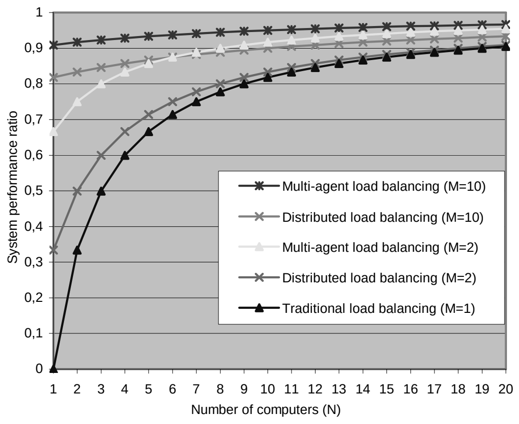


Fig. 8. Ratio of average system performance with one failed computer compared to fully operational system performance

4 EVALUATION AND OBSERVATIONS

Fault-tolerance-related observations and measurements of our multi-agent LB system are presented in Section 4.1; scalability-related observations and measurements are presented in Section 4.2; observed advantages and disadvantages are presented in Section 4.3.

4.1 Fault-Tolerance

The availability of the entire MAS can be increased by increasing the availability of each single agent and/or by defining the strategy for detecting and handling agent failures. Our fault-tolerant strategy was twofold: first, we set up the system-wide monitoring of all agents, and second, we introduced redundancy for all critical-importance agents. While the ATM and the LB Control Agent also utilize redundancy for parallel operation, the Configuration Agent benefits from the active/backup model of operation. Both methods increase the level of fault-tolerance: parallel operation increases the performance while the active/backup model enables the replacement of failed agents.

While periodic checking detects faults in a timely fashion, it does not increase the reliability of a single agent. However, it does increase the reliability and availability of the MAS. Our agent-level fault-recovery policy promptly restarts all the defective agents; the reliability and availability of the MAS is thus increased because not all the agents are active all the time. If a fault is detected when an agent function is not needed, the agent restart does not impact on the operation of the MAS. This kind of error recovery cannot be achieved with traditional programming, where the restart of the main program renders all functionality unavailable.

When increasing the availability of a software system by periodically checking its components, one must find an appropriate trade-off between the check frequencies and the amount of check-imposed load. Higher checking frequencies generate a higher computational load and more network traffic, but they also produce faster failure responses. Additionally, checking frequencies are often restricted by the properties of the checked entities. In our system the different services, agents and servers have different response times. For example, the operating system does not allow the transferring of a virtual IP to another computer in less than approximately 10 s. Consequently, the process of migration of the active ATM lasts approximately 10 s. It is therefore not reasonable to force the ATM check period to be in the sub-second level.

We have compared the presented agent and non-agent versions of a LB system. To test availability of the agent system, we repeatedly terminated various cluster components to test failure response. In the first test we simulated failures by repeatedly terminating ATMs and measured cluster availability. Within one time unit, active ATM was terminated at random time. The test was repeated 100 times for minute, 10 times for hour and one time for day and week period. We used cluster configuration with 2 servers each serving one service and 2 LB computers. Averaged results are presented in Figure 9.

For the second test, we periodically rebooted servers to simulate fatal server errors. Random server was rebooted at random time in each time period. Availability of cluster was measured by connecting every second and counting number of failed requests. LB policy was set to round-robin. The test was run continuously for 100 minutes, 10 hours, one day and one week for each time period respectively. We used cluster configuration with 10 servers each serving one service and 2 LB computers. Averaged results are presented in Figure 10. Note that server bootup process lasted roughly 2 minutes. With one reboot per minute, servers were constantly rebooting, which explains very low cluster availability.

To test the fault-tolerance of both systems we have simulated fatal software errors by randomly terminating software entities and continuously measuring clus-

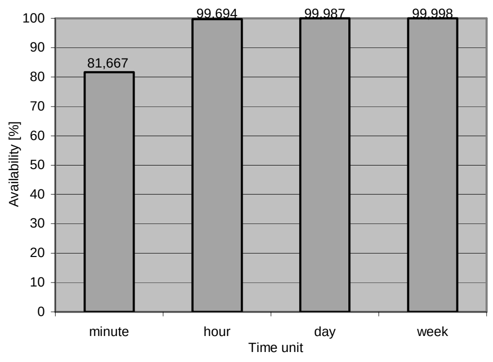


Fig. 9. Availability of the cluster on ATM failure

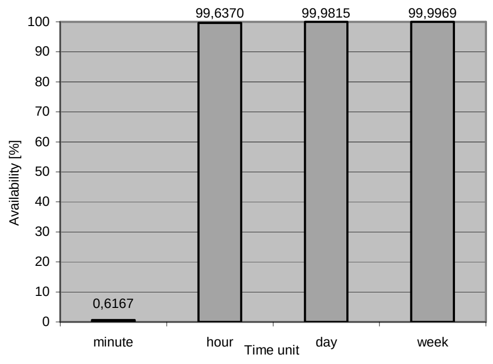


Fig. 10. Availability of the cluster on server failure

ter availability. On each simulated software failure one or more randomly chosen running programs/agents were terminated. The cluster was periodically monitored by issuing a service request every second and counting the number of failed requests. Each test started with a simulated failure and lasted 60 seconds to allow the cluster to finish all its activities. Then another software failure was introduced and so forth, thus simulating continuous failures. Each test was repeated 100 times and the averaged results are presented in Figure 11 and Figure 12.

Both test systems consisted of 2 load balancers and 10 servers, with each serving only an HTTP service. The non-agent version of the LB system utilized 12 software instances, while the agent version utilized 56 agents. As there was a difference between the number of agents and software instances, we tested the robustness of the agent approach by varying the number of simultaneously terminated agents. Since the agents are smaller functional units than the non-agent units, they are consequently smaller and have fewer faults. For non-agent versions each failure was encountered as one software entity while for agent version the error was multiplied (1×, 2×, . . . , 5×) to compensate for larger number of entities in agent version. It should be noted that agents can be restarted without impacting the whole MAS, whereas programs in non-agent systems cannot.

The results presented in Figure 11 and Figure 12 demonstrate both the advantages of the MASs. If an agent-level fault-recovery policy is enabled (Figure 11), which is usually the case, the availability of the MAS with just five terminated agents per test significantly outperforms the traditional version. If the agent-level fault-recovery policy is disabled (Figure 12), then even with three terminated agents per test, the availability of the MAS system is substantially higher than that of the non-agent version. Note that in this case, after ten consecutive tests there were 30 inactive agents, as a result of 30 errors, whereas in the traditional version there were 10 terminated software instances as a result of 10 errors.

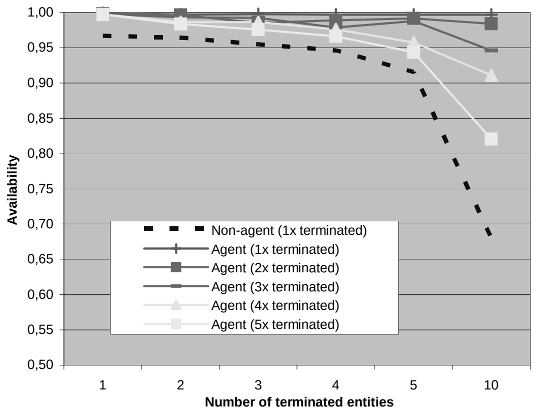


Fig. 11. Availability of the system on software failure with enabled agent fault recovery

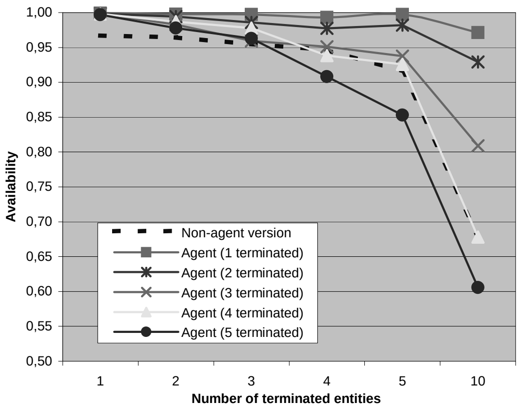


Fig. 12. Availability of the system on software failure without agent fault recovery

Whereas the schematic LB architecture presented in Figure 6 d) could be implemented by non-agent approaches, the internal agent software architecture could not; and although the theoretical analyses presented in Figure 7 and Figure 8 might be attained by advanced, traditional distributed approaches, the practical measurements in Figure 11 and Figure 12 indicate the real advantages of the multi-agent approach over the traditional approach as a result of its internal agent structure.

The multi-agent approach by itself does not necessary increase the level of fault-tolerance or any other property of LB systems. It does, however, enable designers to develop monitoring and error-recovery strategies that can significantly increase availability and reliability, thus increasing the fault-tolerance of the whole LB system.

4.2 Scalability

When designing scalable systems, the most important property of scalable entities is that the computation and communication loads do not scale with the number of entities in the system. One of the major concerns when designing our multi-agent LB system was scalability. As a result, it is scalable in terms of agent infrastructure, agent checking, ATM checking, and agent communication.

However, there are some non-scalable operations regarding servers and services which require agent activities on all servers. It is essential to limit the frequency of these events as network traffic scales linearly with the number of servers and services. These operations are performed only on configuration changes and when checking services and servers. While configuration changes occur only seldom and are expected to last for a longer period of time, all checking activities are performed periodically and must not impact on the system performance. It is therefore necessary to assess the implications of cluster-wide checking.

Check type	An approximate amount of data for one check	Default checking period	Amount of data per 60 s
agent	0.5 Kb	60 s	0.5 Kb
service*	0.2 Kb	30 s	0.4 Kb
server**	0.11 Kb	15 s	0.44 Kb
ATM***	0.5 Kb	1 s	30 Kb

* Check is service dependent. The presented numbers hold for a default HTTP check.
** Check is implementation dependent. The presented numbers hold for default ICMP check.
*** This check is actually a network broadcast of one UDP "heartbeat" packet.

Table 2. Different check types

Although the computational load and the amount of data transferred during each check is reasonably low, one must consider the cumulative effect – a great number of checked entities can significantly increase the network traffic and can thus limit MAS scalability. The check sizes and the default checking periods of the multi-agent LB system are presented in Table 2. Note that the server and service checks

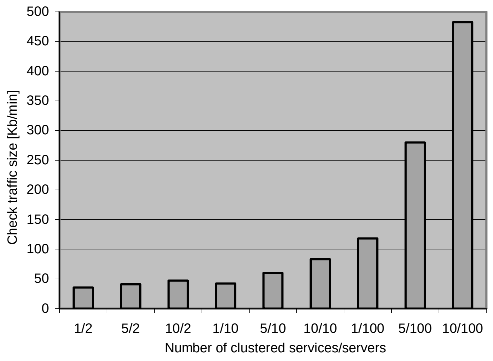


Fig. 13. Cumulative check-related network traffic generated on a LB computer with an active ATM

are performed by corresponding Server/Service Check Agents. We measured the cumulative check-related traffic generated on a LB computer with an active ATM. The cluster configuration was set to 2 LB computers and 2, 10 and 100 servers hosting 1, 5, and 10 services. The results are presented in Figure 13. Our analysis shows that the network traffic is linearly correlated with the number of servers and cannot be easily regarded as insignificant. However, the cumulative values are low and so cluster sizes up to 100 servers are achievable.

Evidently, a MAS is as prone to scalability issues as other distributed approaches. Designers must carefully study possible agent interactions and assess the implications of big agent societies. Normally, scalability issues can be determined at the design phase, and can then be at least reduced, if not completely suppressed.

4.3 Advantages/Disadvantages

It is often hard to estimate the benefits of an agent approach compared to a non-agent approach. New agent systems typically do not have a similar functionality to non-agent systems. This is especially true for large systems, because it is not reasonable to design two systems, one agent-oriented and the other non-agent-oriented. In our case we first designed and implemented the traditional system. The system is in commercial use since 1999 and received several prestigious awards (Corporate IT Best Product winner for enterprise-class customers and Finalist Best Product honor at the Linux Open Source Expo & Conference, and Top Web Solution award at Linux Business Expo held at Comdex Fall '99). Later we designed and implemented functionally similar, but independent agent system having in mind certain experiences with the first system. When comparing both systems, the similarity is sufficient to allow the differences to be overlooked and focus on the key advantages and disadvantages of the agent approach. The comparison between the two systems, i.e., the two approaches, is analyzed through the eyes of designers, developers and users. All three views are summarized in the following subsections.

4.3.1 Designer's View

Comparing the processes of designing a MAS and traditional distributed systems, one can identify many similarities and differences. Apart from using different software-engineering methods, the agent approach incorporates the process of designing distinct roles and agent interactions. This adds an additional layer of abstraction and introduces a more hierarchical and advanced structural design, thereby improving the comprehensibility of the design. One of the biggest advantages was observed in terms of abstraction: agent roles provide new abstract design layers, and agent interaction protocols introduce new abstract-level interfaces. This additional layer greatly improves the overall comprehensibility of the design of a big and complex system.

During the design, we identified the following advantages:

New design abstraction layers. The introduction of agents adds new abstract design layers, which results in a more hierarchical and structural design that is easier to comprehend. This is especially true for large and complex systems, where sheer size prevents a detailed understanding of the design. Numerous smaller and simpler agent designs, defined with agent roles and abstract-level interactions, are easier to comprehend than the design of traditional programs with complex and sometimes hidden connections between the modules. Consequently, the design of single agents can be efficiently shared and split among many designers. This hierarchical structure is also natural for social organizational structures, showing the similarity between people and agent coordination.

It is possible to introduce agents that were not predicted. This is especially important when upgrading the MAS. New requirements are often hard to predict and are commonly based on costumers' feedback. New agents can help improve system performance and introduce new features. Due to the dynamic nature of agent interaction, such new agents seamlessly upgrade the existing MAS to an advanced version.

Decentralization reduces complexity of single software instance. Since there is no central entity, the whole control process is delegated among many agents. Although the interaction between agents increases the complexity, it can be efficiently viewed as an abstract interface to agent functionality. Due to the natural decomposition of tasks and entities, the design of an agent system is thus easier to comprehend. Each agent *per se* is therefore easier to develop than one big program with the functionality of all the agents.

The observed disadvantages are as follows:

It is impossible to predict exactly the system interaction and execution. The complexity of an agent society and unpredictability of system resources makes it impossible to predict exactly the agent interaction and activity over time. In addition, there is no guarantee of system correctness. As both these facts hold true for other distributed approaches as well, they should not be regarded as a strong disadvantage but rather as an inherent property of distributed systems.

4.3.2 Developer's View

The developer's work is often seen as a straightforward process, but the reality is very different. The ingenuity of a developer can significantly speed up the implementation cycle, thus saving time and money; and the appropriate tools make it possible for developers to meet ever tighter deadlines. We argue that the agent-oriented approach is a step in the right direction, as the observed advantages tend to outperform the observed disadvantages.

The advantages are as follows:

The agent environment standardizes the interaction between agents. It helps developers if they can define standard abstract-level interfaces and use standard programming models. With the introduction of agents, an additional conceptual layer increases the comprehensibility of the design. Faster development is a result of an enhanced focus on functionality, and not on the protocols for information/knowledge exchange.

It is possible to increase the fault-tolerance of a MAS. Our measurements confirm that even simple agent-level checking and error-recovery methods improve the fault-tolerance. Due to the inherent property of a MAS that functionality is split among many agents, agents can be efficiently restarted without impacting on the operation of the system. Although the restarting of failed software instances is also possible with the traditional approach, the size of the failed functionality is usually smaller with the agent approach.

The developer must satisfy smaller goals based on local knowledge. Agents are smaller functional units than the non-agent software, and their functionality is strictly limited. Goals are therefore easier to achieve and are always based on local knowledge. Smaller goals also result in reduced local synchronization activities. In general, when comparing only the size of programs and not the complexity of the programs, smaller programs are easier to comprehend than the larger ones. Consequently, if given the same debugging options, they are thus easier to validate and debug.

A MAS forces developers to exploit concurrency. A MAS, as an inherently distributed system, can greatly benefit from exploiting concurrency. The nature of the agent approach forces developers to exploit concurrency. As an example, in our case an agent approach allowed the easy introduction of simultaneous LB, thus improving performance and increasing fault-tolerance.

There may be different solutions to the same problems. With the anticipation of communication-related problems, such as multiple or failed responses, programmers are forced to find different solutions to the same problems. This approach adds robustness to agent systems and helps developers achieve different thinking, which often results in better solutions.

Agents can be reused. If agents are designed in an independent way, they can be reused in other, possibly different agent systems. Clean and independent agent design will improve an agent's reusability, much as good software libraries can be used in many applications.

Developers cooperate on an agent-to-agent interaction basis. Cooperation between developers is not based on a rigid organizational structure. Rather, it is a consequence of the interactions between agents. Developer-to-developer cooperation is a kind of abstraction of agent-to-agent interaction. Such teams are more productive and easier to manage since cooperation is limited to smaller groups, where developers tend to exchange ideas more efficiently.

The disadvantages observed during the development process are as follows:

Developers must predict failed or multiple responses. In the agent environment there is no guarantee about the response to agent queries; there may be one, many or even no response at all. Developers must implement additional safety mechanisms to prevent misinterpretations of an unwanted response, which is often a daunting task.

A MAS introduces new synchronization problems. This is one of the hardest problems when developing distributed systems. The synchronization of multiple activities with multiple requests, where activities and requests can exclude each other, is difficult. In our approach, each agent prevents concurrent access by locking access to its internal state and by serializing mutually exclusive tasks.

4.3.3 User's View

The typical user of a LB system is a system administrator. Although system administrators are technically oriented and above average in terms of accepting technical improvements, this does not automatically mean that they gladly accept any new approach. Novelties are often seen as dangerous, especially in distributed environments. On the other hand, successful new approaches tend to be well accepted, given time.

The most important advantages are as follows:

Robustness of an agent system. The distribution of vital components is an essential factor contributing to greater robustness. A MAS is one of the best representatives of robust distributed systems. Multiple instances of agents can remove single points of failure and so improve performance.

Easier and non-interruptive upgrades. An important property of agent systems is the possibility to shut down agents and restart them. Additionally, agent systems can even operate without some less-important agents. Consequently, one can upgrade or change a system's functionality by changing or adding agents without seriously interrupting the system. For example, in our system, additional LB policies can be easily added with the introduction of new LB Policy Agents.

Users also observed some disadvantages:

Bigger consumption of resources. An increased number of agents and network interactions consumes additional system resources. This is often regarded as a waste of resources. However, it is fair to say that a similar phenomenon was observed with all advanced techniques, e.g., going from go-to to modular and to object-oriented programming. All new techniques demand more and more resources, but enable faster and more straightforward software engineering.

The large number of programs/agents is confusing. Users can get confused by large numbers of agents executing on one or more computers. A typical response is that it should be done in a simpler way. If users are not properly acquainted with agent approaches and agent mentality, or they do not accept it, agent systems can be accompanied with certain resistance.

Agent systems must earn trust. Agents represent a fairly new research field, introducing several new concepts. It is therefore important to present scientific and practical reports on various aspects of agent systems. This paper tries to clarify the advantages and disadvantages of agent approach.

5 RELATED WORK

Most LB- and agent-related papers deal with optimizing LB efficiency using agents. In contrast, we compare the architecture- and design-specific issues of an agent and non-agent version of a LB system. As Chow and Kwok [3] point out, very little published research has been done on the load-balancing aspect to capture the essence of agent systems in such a distributed environment.

Cao et al. [2] analyze a mobile-agent approach to load balancing. They propose a framework that uses mobile agents to implement scalable load balancing on distributed web servers. Their comparison with the traditional message-passing LB methods shows that the mobile-agent-based approaches exhibit the merits of high flexibility, low network traffic and high asynchrony. Schaerf et al. [22] study the process of multi-agent reinforcement learning in the context of load balancing in a distributed system. They define a precise framework, called a multi-agent multi-resource stochastic system, which involves a set of agents, a set of resources, probabilistically changing resource capabilities, the probabilistic assignment of new jobs to agents, and probabilistic job sizes. Their analysis of adaptive load balancing demonstrates that adaptive behavior is useful for efficient load balancing. They define a pair of parameters that affect that efficiency in a non-trivial fashion and show that the naive use of communication might not improve and could even deteriorate the system efficiency. Their research focuses primarily on efficient LB, whereas we analyze different LB architectures. The paper of Givas and

Turner [7] surveys LB using agents. Although their analysis of centralized and distributed approaches to agent-oriented LB is rather basic, they conclude that agents offer a novel, dynamic and unlimited approach. Chow and Kwok [3] outline the space of LB design choices in the arena of multi-agent computing. They also propose a novel communication-based LB algorithm together with its evaluation. The proposed algorithm assigns a credit value to each agent, depending on its affinity to a machine, its current workload, its communication behavior, its mobility, etc. The paper focuses on optimizing the LB algorithm using agents, whereas our paper tries to address other essential agent-related issues, such as agent-oriented design, scalability, and the fault-tolerance of agent-based LB systems.

A large number of techniques for fault-tolerance can be found in the literature relating to traditional databases and distributed systems. Most of these recovery methods [1] focus on replication techniques that permit critical system data and services to be duplicated as a way of increasing reliability. There also exist fault-tolerant middleware frameworks providing transparent fault-tolerance for enterprise applications. One such, designed for CORBA applications, is described in [18]. However, our paper focuses on agent-oriented approaches. Hägg [8] uses external sentinel agents that listen to all broadcast communication, interact with other agents and use timers to detect agent crashes and communication-link failures. The sentinels in Hgg's approach analyze all the communication going on in the MAS to detect state inconsistencies. However, this approach is not realistic for systems with a high volume and message frequency. Klein [14] proposes the use of an exception-handling service to monitor the overall progress of a MAS. Here, agents register a model of their normative behavior with the exceptional-handling service that generates sentinels to guard the possible error modes. Such an exception-handling service is also a centralized approach, which is not suitable for scalable distributed systems.

It is important to note that no related work is known to the authors showing principal advantages of the multi-agent approach for LB, and no known publication comparing a large implemented traditional LB systems and an agent-based one performing the same task thus enabling a thorough practical and theoretical comparison.

6 CONCLUSION

We have analyzed a non-agent and an agent-based LB approach and the corresponding fully implemented systems, each consisting of several 100.000 lines of source code. The two systems perform the same LB functions and basically differ only in their architecture. This gives us a unique opportunity to practically and theoretically compare the two different approaches. The multi-agent architecture of LB systems in theory introduces important improvements, such as better average performance when one computer is not working and a lower system-error probability. In terms of the development process, fault-tolerance, and scalability, the agent approach offered the expected improvements, both in objective real-world measurements and in the subjective observations of designers, developers and users.

On the other hand, we could not overcome several well-known problems when designing distributed systems. For example, handling failed entities, synchronization problems, and query-response-related issues turned out to be the same as in any distributed programming.

It is important to be aware of the advantages and disadvantages of the agent and non-agent approaches, but the most important point is whether the advantages prevail. For LB systems, our theoretical analysis and practical experiences both indicate that the advantages of multi-agent LB systems clearly outweigh the observed disadvantages.

Acknowledgment

This work was supported in part by the Slovene Ministry of Education, Science and Sport, and in part by TurboLinux Inc. and Active Tools, d. o. o.

REFERENCES

[1] Birman, K. P., editor: Building Secure and Reliable Network Applications. Part III, Reliable Distributed Computing. Chapters 12-26. 1996.

[2] Cao, J.—Sun, Y.—Wang, X.—Das, S. K.: Scalable Load Balancing on Distributed Web Servers Using Mobile Agents. Journal of Parallel and Distributed Computing, Vol. 63, 2003, No. 10, pp. 996–1005.

[3] Chow, K. P.—Kwok, Y. K.: On Load Balancing for Distributed Multiagent Computing. IEEE Transaction on Parallel and Distributed Systems, Vol. 13, 2002, No. 8, pp. 787–801.

[4] Comer, D. E.: The Internet Book: Everything You Need to Know about Computer Networking and How the Internet Works. Prentice Hall, 3rd Edition, 2000.

[5] Durfee, E. H.—Lesser, V. R.—Corkill, D. D.: Trends in Cooperative Distributed Problem Solving. In IEEE Transactions on Knowledge and Data Engineering, Vol. 1, 1989; No. 1, pp. 63–83.

[6] Gray, R: Agent Tcl: A Flexible and Secure Mobile-Agent System. In Proceedings of the Fourth Annual Tcl/Tk Workshop, 1996.

[7] Grivas, M.—Turner, S. J.: Agent Technology in Load Balancing for Network Applications. In Proceedings of the Workshop on Intelligent Agents, 1998.

[8] Hägg, S.: A Sentinel Approach to Fault Handling in Multi-Agent Systems. In Proceedings of the 2nd Australian Workshop on Distributed AI, Cairns, Australia, 1997.

[9] Iglesias, C. A.—Garijo, M.—Gonzalez, J. C.: A Survey of Agent-Oriented Methodologies. In Intelligent Agents V, Proceedings of the ATAL-98, Springer-Verlag, 1999.

[10] Kernighan, B. W.—Pike, R.: The Practice of Programming. Addison-Wesley, 1999.

[11] Koutny, T.—Safarik, J.: Load-Balancing Using Autonomous Co-Operating Nodes. Proceedings of Mobile Future and Symposium on Trends in Communications, SympoTIC '03., pp. 153–155, 2003.

[12] Jennings, N. R.: On Agent-Based Software Engineering. Artificial Intelligence, Elsevier, Vol. 117, 2000, pp. 277–296.

[13] Jennings, N. R.—Bussmann, S.: Agent-Based Control Systems. IEEE Control Systems Magazine, Vol. 23, 2003, No. 3, pp. 61–74.

[14] Klein, M.—Dellarocas, C.: Exception Handling in Agent Systems. In Proceedings of the Third Annual Conference on Autonomous Agents Autonomous Agents, Seattle, 1999.

[15] Lind, J.: A Development Method for Multiagent Systems. In Cybernetics and Systems: Proceedings of the 15th European Meeting on Cybernetics and Systems Research, 2000.

[16] Lúčny, A.: Building Complex Systems with Agent-Space Architecture. Computing and Informatics, Vol. 23, 2004, No. 1.

[17] Návrat, P.—Manolopoulos, Y.—Vossen, G.: Special Issue on ADBIS 2002: Advances in Databases and Information Systems. Information Systems, Vol. 29, 2004, No. 6, pp. 437–438.

[18] Narasimhan, P.—Moser, L. E.—Melliar-Smith, P. M.: Eternal – A Component-Based Framework for Transparent Fault-Tolerant CORBA. Software Practice and Experience, Vol. 32, No. 8, pp. 771–788.

[19] Nwana, H.—Ndumu, D.—Lee, L.—Collis, J.: ZEUS: A Toolkit for Building Distributed Multi-Agent Systems. In Applied Artificial Intelligence Journal, Vol. 13, 1999, Nos. 1/2, pp. 129–185.

[20] Pinho, L. M.—Vasques, F.—Wellings, A. J.: Replication Management in Reliable Real-Time Systems. In Real-Time Systems, 2004, pp. 261–296.

[21] Pnueli, A.: Specification and Development of Reactive Systems. In Information Processing 86, Elsevier/North Holland, 1986.

[22] Schaerf, A.—Shoham, Y.—Tennenholtz, M.: Adaptive Load Balancing: A Study in Multiagent Learning. In Journal of Artificial Intelligence Research, 1995 No. 2, pp. 475–500.

[23] Stone, P.: Multiagent Systems: A Survey from a Machine Learning Perspective. Autonomous Robots, Vol. 8, 2000, No. 3.

[24] Tanenbaum, A. S.—van Steen, M.: Distributed Systems: Principles and Paradigms. Prentice-Hall, 2002.

[25] Verissimo, P.—Kopetz, H.: Design of Distributed Real-Time Systems. In Shape Mullender, editor, Distributed Systems, Chapter 19. Addison-Wesley, 2nd edition, 1995.

[26] Wooldridge, M.—Jennings, N. R.: Software Engineering with Agents: Pitfalls and Pratfalls. In IEEE Internet Computing, Vol. 3, 1999, No. 3.

[27] Zambonelli, F.—Jennings, N. R.—Omicini, A.—Wooldridge, M.: Agent-Oriented Software Engineering for Internet Applications. In Coordination of Internet Agents: Models, Technologies and Applications, Springer, 2001.

Andraz Bezek received his M. Sc. degree in artificial intelligence from Faculty of Computer and Information Science of University of Ljubljana in 2002. His current position is research assistant at the Department of Intelligent Systems at the Jozef Stefan Institute, Ljubljana. He is finishing a Ph. D. study in computer science at the Faculty of Computer and Information Science of University of Ljubljana. His research interests include multi-agent systems, agent oriented software engineering, multi-agent strategy modeling, and machine learning.

Matjaz Gams is professor of computer science and informatics at the Ljubljana University and researcher at the Jozef Stefan Institute, Ljubljana, Slovenia. He teaches several courses in computer sciences at graduate and postgraduate level at Faculty of Computer and Information Science, Faculty of Economics, and several others. His research interests include artificial intelligence, intelligent systems, intelligent agents, machine learning, cognitive sciences, and information society. His publication list includes over 250 items, 50 of them in established scientific journals. He is an executive contact editor of the Informatica journal and editor of several international journals. He is head of the Department of Intelligent Systems, cofounder of the Engineering Academy of Slovenia, Artificial Intelligence Society, and Cognitive Sciences Society in Slovenia; currently he is vice-president of ACM Slovenia and secretary of the Engineering Academy of Slovenia.