Increasing Fault-Tolerance of Multi-Agent Systems

Andraz Bezek, Matjaz Gams
Jozef Stefan Institute, Jamova 39, 1000 Ljubljana, Slovenia
{andraz.bezek, matjaz.gams}@ijs.si

Keywords: fault-tolerance, multi-agent systems

Received: June 15, 2003

We analyze fault tolerance in load balancing systems. First, we theoretically analyze some aspects of fault-tolerance of traditional and multi-agent approaches. Second, we investigate efficient fault detection showing that multi-agent systems can increase fault-tolerance by improving fault detection of failed single agents. The presented ideas are applied in a multi-agent system for fault-tolerant network load balancing. A detailed description of fault-tolerant issues in design is also given. Finally, the paper presents evaluation of a multi-agent application for network load balancing.

1 Introduction

Designing and building high quality industrial-strength software is difficult, and it has been claimed that such development projects range among the most complex construction tasks. Among software projects, building reliable large Internet services is one of very difficult tasks especially because virtually continuous uptime and consistent response time are crucial. Stable services must be able to cope with many undesired factors such as explosive growth of traffic over the Internet, and possible hardware and software failures.

A wide range of software engineering paradigms has been devised, and each successive development claims either to make the engineering process easier or to extend the complexity of applications that can feasibly be built. Although there is reasonable evidence to support these claims, researchers continually strive for more efficient and powerful software engineering techniques. Recently, multi-agent systems received a lot of attention as a potential mainstream initiative for distributed software engineering [5]. 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 [2]. Recently, the term MAS has been given a more general meaning, and it is now used for all types of systems composed of multiple autonomous agents showing the following characteristics:

- each agent has incomplete capabilities to solve a problem,
- there is no global system control,
- data is decentralized and
- computation is asynchronous.

MAS is a distributed reactive system, which maintains an ongoing interaction with environment. It has long been recognized that reactive systems are among the most complex types of systems to design and implement [9]. Great effort has been devoted to developing software tools, programming languages, and methodologies for managing this complexity – with some success. But for certain types of reactive systems, even specialized software engineering techniques and tools fail. According to Jennings et al. [4], one can broadly subdivide these systems into three classes:

- open systems,
- complex systems and
- ubiquitous computing systems.

An *open system* is capable of dynamically changing its own structure. Consequently its components can be heterogeneous and added dynamically. They also can be changed over time using different software tools and techniques.

Development of *complex systems* requires efficient tools for handling software complexity. The most powerful concepts are modularity and abstraction. Namely, if a problem domain is particularly complex, large, or unpredictable, then it may be reasonable to develop a number of distinct modular components. An agent-oriented approach is a powerful alternative for making systems modular. In case any interdependent problems arise the agents in the systems must cooperate with one another to ensure that interdependencies are properly managed. In such domains an agent-based approach means that the overall problem can be partitioned into a number of smaller and simpler components, which are easier to develop and maintain, and which are specialized at solving the constituent sub-problems. This decomposition allows each agent to employ the most appropriate paradigm for solving its particular problem, rather than being forced to adopt a common uniform approach that represents a compromise for the entire system, which is not optimal for any of its subparts. The notion of an autonomous agent also provides a useful abstraction in the same way that procedures, abstract data types, and objects provide abstractions.

Ubiquitous systems are expected to ease the interaction between humans and computers. But today the user of a software product typically has to describe each step that needs to be performed to solve a problem down to the smallest level of detail. Ubiquitous system should be able to recognize opportunities and act in such a way to help the users to achieve their goals. One can think of such

systems as assistant agents, capable of acting with the user in order to achieve the user's goals.
In MAS the locus of control is dispersed among agents which coordinate and plan their actions according to their local perception of environment. If we try to improve fault-tolerance of a MAS, we must improve fault-tolerance of all agents within MAS, or provide system-wide methods to enable fault detection and recovery. This paper addresses the latter problem. It overviews fault-tolerance with agent systems and presents a fault-detection models together with a fault-tolerant agent application for network load balancing.
The structure of this paper is as follows: In Section 2, we present functional characteristics and differences between traditional and multi-agent load balancing system. Section 3 deals with fault tolerance in agent societies. An analysis of fault-tolerance related differences for two load balancing systems is presented in Section 3.1. Fault detection models are presented in 3.2 and survey of related work is given in Section 3.3. A fault tolerant multi-agent system is presented in Section 4. Finally, conclusions are presented in Section 5 by reviewing the ideas presented in the paper.

2 Traditional and Multi-Agent Load Balancing

We refer to the term *load balancing* (LB for short) as network load balancing or - more precisely - as IP-level load balancing. The term cluster refers to all computers within a load balancing system, thus including all servers, load balancers and 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/load balancing functionality.
The purpose of network LB system is to evenly distribute incoming network traffic among servers in a cluster. The distribution is done according to desired LB policy, which often takes into account performance metrics such as network traffic or processor load.

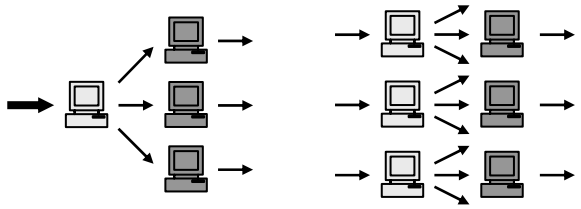

Figure 1: Network data flow for traditional (a) and multi-agent (b) load balancing systems.

A *traditional load balancing system* consists of a single load balancer and a set of servers as presented in Figure 1, a). Accordingly, the input flow is balanced among servers and resulting output traffic is redirected directly to users. Beside obvious performance benefits, a design like this also increases fault tolerance. The failed server can easily be replaced by redirecting its traffic destined among other active servers. Server and service failures are detected by periodic server/service checking. *Multi-agent load balancing systems*, shown in Figure 1, b), enhance scalability by employing an arbitrary number of load balancers. They require a preceding load balancer which distributes input traffic between distributed LB systems. Load balancing at this level can employ coarser distribution policy without any impact on underlying systems. Most often round-robin DNS solution is used to assure geographical distribution of traffic. The main benefit is the possibility to change the size of LB cluster. The LB software can be positioned at any location. Load balancers and servers can be distributed across the Internet, most often geographically. This design makes fast fault detection and recovery possible by enabling LB agents to constantly monitor system activities. The main benefit compared to the previous approach is the fact that servers behind the failed load balancer can be still used by other balancers. A multi-agent approach may utilize an arbitrary number of load balancers and servers.

3 Fault-Tolerance in Agent Societies

In theory, fault tolerant systems (chapter 7. in [7]) should not have a single point of failure and should be resistant to any failure, including the hardware and software ones. In reality we aim to achieve sufficiently high degree of fault tolerance. According to [10], fault tolerant services must be designed to achieve high availability, safety, maintainability, and reliability. Availability is defined by the percentage of time during which 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. Although web services do not fall into the same safety category as, for example, nuclear power plants, one can easily understand the importance of safety considerations when designing web-based services. 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 property that a system can run continuously without failure. If a system goes down for one millisecond every hour, it has an availability of over 99.9999 percent, but it is still highly unreliable. Similarly a system that never crashes but is shut down for a week every year has high reliability but only 98 percent availability.
When developing MASs developers often overlook the importance of fault-tolerant software development. Sometimes, especially in *open systems*, it is impossible to enforce fault-tolerant developing strategies since we do not control the development process. Designers can only enforce fault-tolerant strategies through selected multi-agent architecture design. *Complex systems* are by definition difficult to comprehend and are often affected by hard-to-spot faults. Sheer size of such systems prevents complete and systematic testing of system functionality. System-wide fault-tolerant models are

therefore welcome as additional mechanisms, which increase fault resistance of a MAS. Although one can argue that faults in *ubiquitous systems* are least harmful, we state that each fault can affect the very essence of such systems.

3.1 Fault-Tolerance in Load Balancing Systems

We compare LB systems presented in Figure 1 in terms of reliability. If p is the computer error probability over a given time, M is a number of load balancers, and N a number of servers within the system, then we can estimate overall error probabilities. If we assume instantaneous error detection and response, then error probability is as follows:

- Traditional load balancing system with N servers:

$$p + (1 - p) \cdot p^N$$

- Multi-agent load balancing system with N servers and M load-balancers:

$$p^M + p^N - p^{N+M}$$

Results in Figure 2 show that the traditional load balancing system is substantially less reliable than the multi-agent versions for M=2 and N=10. For small error probability p and large numbers of N and M, multi-agent systems become highly reliable.

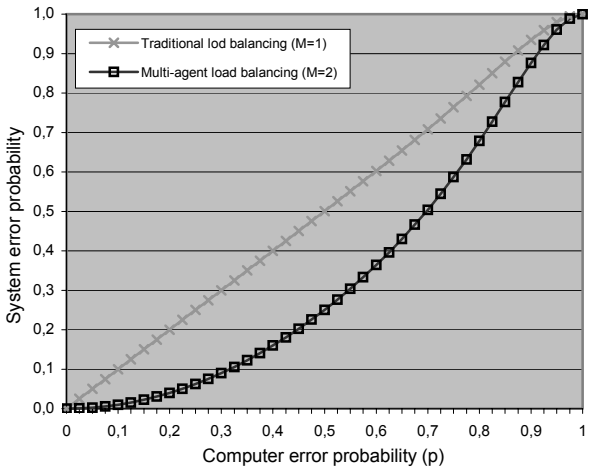

Figure 2: System error probability for M=2 and N=10.

The assumption in previous analysis is that a failed computer is never repaired. This situation is reasonable when we analyze error probability over short intervals of time, e.g. one hour. For longer time periods computers are repaired. If p is the probability of an error over time t and r is average repair time, then the probability that a computer is not operational is $p \cdot r/t$. Most common situation in practice is that either the system is fully operational or one computer is not working. In the latter case, we assumed that all computers have the same failure probability and that the system performance is proportional to the number of working servers. Now we can analyze the ratio of average performance between a system with one failed computer and a system with all working computers for various approaches:

- Traditional load balancing system with N servers:

$$\frac{N\,(N-1)}{(N+1)\,N}$$

- Multi-agent load balancing system:

$$\frac{M}{(N+M)} + \frac{N\,(N-1)}{(N+M)\,N}$$

Multi-agent LB systems shown in Figure 1 b), perform substantially better with one computer down than the traditional version as shown in

Figure 3.

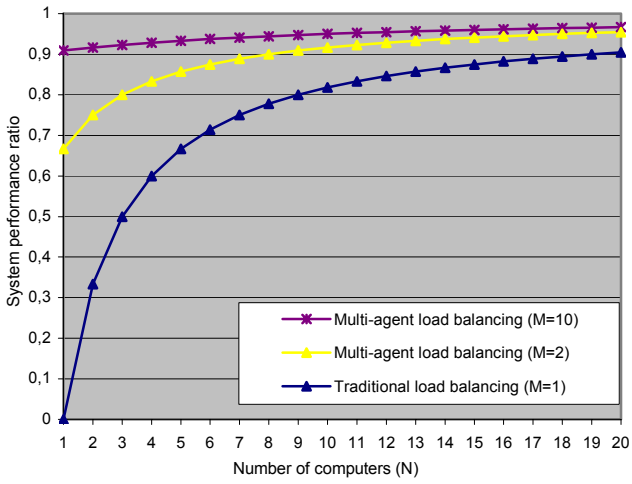

Figure 3: Ratio of average system performance with one failed computer compared to system with all working computers.

Multi-agent systems are prone to failures as any distributed system. Agents and resources may become unavailable due to machine crashes, communication breakdowns, process failures and numerous other hardware and software failures. Most of the work done in fault handling for multi-agent systems 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 MAS requires special infrastructural support, such as support for continuous and fault-tolerant agent communication.

3.2 Fault Detection Models

Software faults can be avoided by appropriate software development process. But often, an open agent society does not allow us to enforce software merits on participating agents. In addition, software and hardware

faults are invariant property of all systems, both software and hardware ones. In order to increase fault-tolerance of an agent system, we must introduce a system-wide strategy to detect faults and to take appropriate steps to reduce harmful consequences. The first step therefore is to detect software faults. Most often we want to keep the fault-detection time below some reasonable time limit. To enforce such limit, faults can be detected in a timely manner by periodical checking. The proper check frequency ensures fault-detection within reasonable limits. As faults can severely affect agent functionality, we cannot assume that agents will detect their own faults. They must be detected by some other entities, which raises another interesting problem. How can this entity – call it a check agent – detect faulty operation of other agents? We present some criteria to decide what kind of check models a designer can utilize. The first one is based on check semantics. Accordingly, a check can belong to different semantic levels:

- *Keep-alive check*: a basic query which inspects checked entity whether it is alive or not. Most often this check can be performed without agent knowledge (e.g. system call for process status).
- *Semantic-free check*: a query asks agents to report its status. The task of performing check is delegated to agent where response is only "good" or "bad". Lack of response also indicates agent's fault.
- *Semantic-full check*: a series of queries which test agent functionality. The queries are actually requests for agent functionality. Agent response is parsed and analyzed for proper operation. Check agent must therefore be able to understand the semantics of response.

As each higher semantic level supersedes the lower one in terms of check efficiency, it is often not possible to implement the full semantic check. The functionality of a checked agent can be unknown or simply too complex to implement. It is therefore a designer's obligation to devise appropriate level of check semantics.

Another criterion deals with entity which performs a check. Theoretically, it can be the agent itself. Although possible, such check does not work for a majority of faults as they can affect the operation of an agent. As the sole other possibility, different agent instance must perform a check. However, we can distinguish between different check models, as shown in Figure 4:

- *Buddy*: each agent is checked by it's own check entity.
- *Star*: a group of agents share the same check entity.
- *Parallel*: each agent in a group checks all other ones.

Table 1: Various statistics for different check models.

Check model	Buddy	Star	Parallel
Check channels	n	n	n(n-1)/2
Additional check entities	n	1	0
All entities	2n	n+1	n
Different check entities	n	1	1

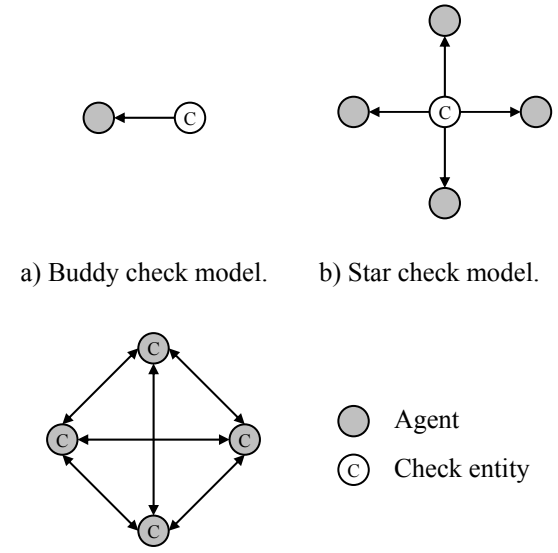

Figure 4: Different check models in multi-agent systems.

Buddy model is the most appropriate when checking a single agent instance with unique functionality within MAS. The downside of this model is a big number of checking entities which is the same as number of agents. *Star model* is efficient where we have a group of agent instances with the same functionality. The same functionality is checked for all agent instances. Obvious advantages are single check entity and efficient resource allocation. The downside is a possible failure of a check entity which suppresses all further check activities. P*arallel model* solves this problem by assigning task of checking to agents themselves; if one agent fails, all others will still perform checking. As agents check each other, they do not need additional check entity. Accordingly their functionality is extended with the check process. This fact also eases the problem of semantic-full check as developers already know the exact functionality of the agent and can therefore design efficient check queries and response analyzers. As a negative side of this model one must mention increased communication-related resources as agents connect with each other. Communication channels must employ efficient delivery methods, e.g.: broadcast on local networks and multicast on wide area networks. Analysis of each model is presented in Table 1, where n represents the number of agents within MAS.

3.3 Related Work

A large number of techniques for fault-tolerance can be found in the traditional database and distributed systems literature. Most of these recovery methods [1] primarily focus on replication techniques that permit critical system data and services to be duplicated as a way to

increase reliability. However, this paper focuses on agent-oriented approaches. Jennings [4] showed that as the world becomes more complex and variable and plans tend to fail more often, agent teams as a whole waste fewer resources and are more robust than self-interested agents. This approach is similar to ours in that both approaches are based on the theory of teamwork. However, we explicitly address the problem of fault-tolerance whereas Jenning's work is more focused towards cooperative problem solving. Kumar and Cohen [8] argued that teamwork might be used to create a robust brokered agent architecture that can recover a MAS from broker failures without incurring undue overheads. Their brokered architecture also guaranteed a specified number of brokers in a large MAS, where agent autonomy can be used to guarantee acceptable levels of quality of service by an agent. Hägg [3] uses external sentinel agents which listen to all broadcast communication, interact with other agents and use timers to detect agent crashes and communication link failures. The sentinels in Hägg's approach analyze the entire communication going on in the MAS to detect state inconsistencies. However, this approach is not realistic for systems with high volume and message frequency. Klein [6] proposes to use exception-handling service to monitor the overall progress of a MAS. Agents register a model of their normative behavior with the exceptional-handling service that generates sentinels to guard the possible error modes. Such exception handling service is also a centralized approach, which is not suitable for scalable distributed systems.

4 A Fault-Tolerant Multi-Agent System

In this section we briefly present a fault-tolerant multi-agent LB system. Its architecture is presented in Figure 1 b). We designed 12 different agent classes. Depending on the agent class, an agent instance can reside on a server, load balancer, management computer, or on each computer within a cluster. A schematic diagram for agent connections is presented in Figure 5. Server-based agents are responsible for handling service and server-related activities: full service/server administration, gathering of service/server-related statistics, and synchronization of service data. Agents hosted on LB computers are LB-oriented and cluster-oriented ones. They perform important activities, such as: checking activities regarding services, servers and load-balancers together with the control of LB software, and enforcement of desired LB policy. Agents with management status handle various cluster-related activities. They provide global configuration together with user interface, report errors, and perform monitoring of agents. Table 2 presents all agent classes with their names, optional acronyms in parenthesis and locations - all in the first line and short descriptions afterwards.

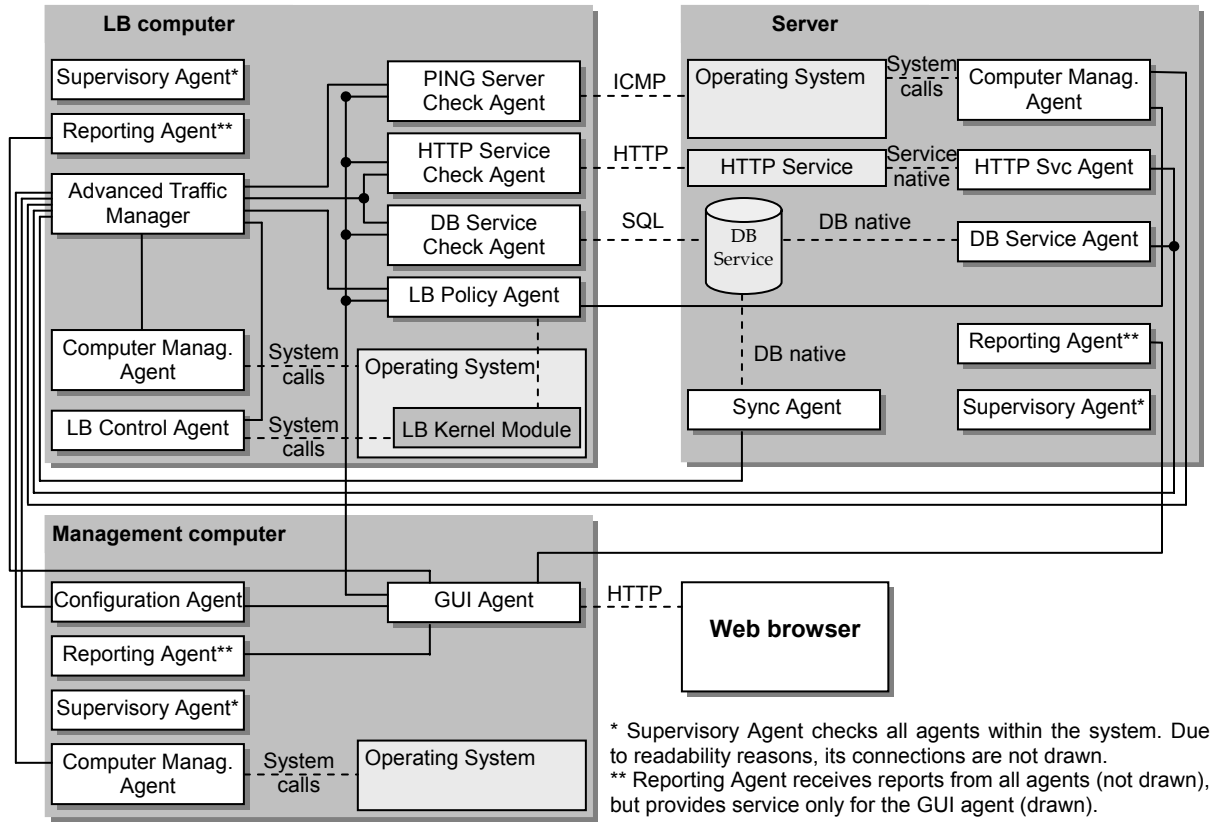

Figure 5: Agent interdependencies and structure of multi-agent LB system.

Table 2: Descriptions of agents in a multi-agent load balancing system.

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 LB software according to configuration, assigns servers and services to LB software, and periodically checks their operation. In case of failure, the server or service is immediately marked as inactive.
Service Management Agent (SMA), server based. A service-specific agent, which performs all service-dependent administration. Its primary task is to setup, 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 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 number of network connections on servers) and compute weights to distribute traffic appropriately.
LB Control Agent (LBA), LB based. Acts as a translator to LB software translating agent commands to 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 amount of free disk space).
Server Check Agent, LB based. Checks if a server is working properly. It is possible to implement several different server checks.
Service Check Agent, LB based. Checks if a service is working properly. Each service demands a different instance of service check agent.
Configuration Agent, management based. Keeps configuration and controls simultaneous access to it. It also broadcasts all recent configuration changes to the agent system.
GUI Agent, management based. Acts like a http server for users, and an intermediate agent to the agent system. Its task is to provide 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 role defined in configuration it must start or stop agent operation of each computer within cluster. In order to deal with unexpected software errors, a special agent was assigned to periodically check the health of running agents. In case of no response, the failed agent is forcefully terminated and restarted.
Reporting Agent (RA), cluster based. Reports various messages and errors to user (via GUI agent).

4.1 Fault-Tolerant Design Issues

Since the fault-tolerant computing paradigm expects failures as a rule and not as an exception, the system must be able to reasonably cope with agent failures. To systematize fault-related activities we introduced different importance levels together with its fault-tolerant design principles. Table 3 presents all importance levels together with corresponding agent classes. Consequently, our design considers the following four levels of agent importance:

- **Core agents** perform tasks essential for proper functioning of the system which stops or is operating erroneous without them. An example for that would be the Advanced Traffic Manager that controls vital cluster management activities.
- **Support agents** carry out tasks regarding management of services, servers and agents. They are needed only for startup and shutdown activities or for reconfiguration of a cluster. For example, the lack of Service Management Agent would prevent the cluster to start or stop certain service but the system would remain stable.
- **Regulative agents** perform partial cluster optimization, and are not critically required to run the system. For example, all checking agents are optional; the system is working also without them. However, the system is thus unable to detect failures of servers or services.
- **User agents** are needed only for users to access cluster management. Its inexistence does not impact the operation of the system.

Table 3: Agent classes with corresponding importance levels. Different levels of shading illustrate importance levels.

Core agents (Critical importance)	**Support agents** (High importance)	**Regulative agents** (Medium importance)	**User agents** (Low importance)
Advanced Traffic Manager	Computer Management Agent	LB Policy Agent	GUI Agent
LB Control Agent	Service Management Agent	Server Check Agent	Reporting Agent
Configuration Agent	Supervisory Agent	Service Check Agent	
	Synchronization Agent		

Table 4: Actions following possible problems

Entity	Possible problems	Action	Semantic level	Check model
agent	agent crash agent malfunction	restart agent terminate & restart agent	Semantic-less	Star
service*	service not accessible service crash service malfunction	try to setup appropriate network interface restart service exclude it from operation	Semantic-full	Star
ATM	computer crash LB software malfunction	exclude it from operation exclude it from operation	Keep-alive	Parallel
server**	computer crash server malfunction	exclude it from cluster force server shutdown and exclude it from cluster	Keep-alive and Semantic-full	Star

* Each reported problem forces service on a computer to be excluded from cluster. It is included later if service checks report that action was successful.

** Each server with reported problems is excluded from cluster. It is included later if server checks report that action was successful.

To increase the fault-tolerance of our system, we incorporated failure prevention for different entities within MAS. The anticipated actions on possible problems together with their semantic level and check model are summarized in Table 4.

Instances of Supervisory Agent class perform basic agent monitoring. Its main function is to monitor agents for proper operation. Since implementing semantic-full check for arbitrary agent is too time consuming and agents can spawn several processes thus forbidding keep-alive queries with system process routines we implemented semantic-less checking. On each check query, agents respond with their state. An agent that does not respond to a check query is considered failed. Because of relatively high number of running agent instances, we designed a distributed version of star check model. With it one instance Supervisory Agent reside on each computer within MAS and monitors only local agents; i.e. it checks all agent instances residing on the same computer. A failure of a single Supervisory Agents therefore only affects monitoring activities on one computer. Service, server, and ATM failure detection is also achieved by periodical checking of each entity. Since the number of services and servers is arbitrary high we chose a check model with the lowest communication-related overhead: a star check model. All checking is performed on an active load-balancer by Service/Server Check Agents, as shown in Figure 5. Servers can be checked with ICMP keep-alive queries (i.e. ping) or semantic-full queries utilizing Computer Management Agent hosted on servers. All services are checked with semantic-full queries (e.g. we implemented HTML queries for checking web services). The most important agent within our system, ATM, was designed with a special protocol to enable mutual checking of ATMs. According to parallel model, each ATM checks other active ATMs. Active ATMs announce their active presence by periodically broadcasting "heartbeat" messages over LAN that also replaced $O(n^2)$ communication channel utilization with only O(1). If, for some known period of time, this message is not received by other ATMs, the sender is considered failed. In this case, elections start and the newly elected ATM begin acting as a primary, while previous active ATM is demoted. Our design allows arbitrary number of ATMs to act as primary while other act as backup ones. Each ATM owns its unique identification number (ID), while each possible active position is represented as one slot and backup ATMs take part in elections for each slot separately. Active ATM for one slot cannot participate in elections for another slot. The voting protocol for each slot is described in Figure 6.

When ATM is elected as active, it starts advertising virtual IP of a running cluster. In LAN environments this can be achieved by advocating virtual IP number with technique called ARP spoofing, which involves constructing forged ARP replies. With this technique we can convince the network gateway that IP of nonexistent computer is actually owned by an existent computer – a load balancer in our case. This enables us to have a virtual IP number, which is associated with an active LB. The traffic destined to virtual IP number is redirected to active ATM which hosts operating LB software.

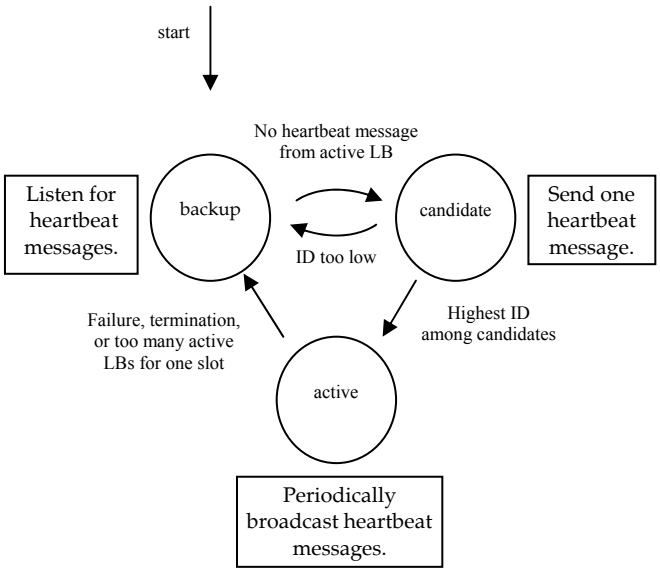

Figure 6: State diagram for voting protocol used in elections for the active load balancer.

The design of Configuration Agent proved to be quite demanding. 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 known implications of such decision, we softened parallelism requirements. The final design allowed only one active Configuration Agent, while others are running in a backup mode. To achieve configuration consistency, all configuration changes can only be committed to the active instance. All backup instances synchronize its configuration with the active one. In case that the active agent fails, configuration can still be retrieved from the backup ones. The location of the active instance is left to the system administrator so as to allow full control over the current configuration computer.

The separated design of agent classes and further separated implementation of agent instances introduces one important improvement over traditional programming. With the latter, developers must corporately develop a big and complex program. Although its design can be modular and developers develop distinct modules, it poses several implementation-related problems. For example, as developers introduce errors, the development process is stopped for all developers working on the same program. Sometimes, the big and complex structure prohibits deep understanding causing developers to overlook hidden module dependencies and introduce hard-to-detect bugs. In the development of MAS developers are more evenly distributed on development of separate agent instances, which also tend to be smaller implementation tasks. Typically one developer develops the whole agent, thus removing inefficiencies presented with the traditional programming. Consequently, the development process is more straightforward, thus faster and less error-prone thus also more fault-tolerant.

5 Conclusion

We have theoretically analyzed fault-tolerance for traditional and multi-agent load balancing systems. Our analysis show that multi-agent load balancing systems formally introduce important improvements such as lower system error probability and better average performance in case one computer is not working. We also presented different semantic check levels and check models. Our analysis reports applicability conditions for different semantic levels and models. The presented ideas are then described in an application of multi-agent load balancing system. We show that presented models do not increase fault-tolerance of single agent instance but clearly improve fault-tolerance of a whole MAS.

It is important to be aware of the advantages and disadvantages of agent and non-agent approaches, but the most important is whether advantages prevail. For load balancing, our theoretical analysis and practical experiences both indicate that advantages of MAS LB systems evidently overweight observed disadvantages.

References

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

[2] E. H. Durfee, V. R.Lesser and D. D. Corkill, "Trends in Cooperative Distributed Problem Solving," in IEEE Transactions on Knowledge and Data Engineering, KDE-1(1), pp. 63-83, 1989.

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

[4] N. R. Jennings, "Controlling Cooperative Problem Solving in Industrial Multi-Agent Systems using Joint Intentions," Artificial Intelligence. 75(2), pages 195-240, 1995.

[5] N. R. Jennings, "On agent-based software engineering," Artificial Intelligence 117, pp. 277-296, Elsevier, 2000.

[6] M. Klein and C. Dellarocas, "Exception Handling in Agent Systems," Autonomous Agents '99, Seattle, 1999.

[7] S. Tanenbaum, and M. van Steen, "Distributed Systems: Principles and Paradigms," Prentice-Hall, 2002.

[8] S. Kumar and P. R. Cohen, "Towards a Fault-Tolerant Multi-Agent System Architecture,", in Proceedings of The Fourth International Conference on Autonomous Agents, 2000.

[9] A. Pnueli, "Specification and Development of Reactive Systems," in Information Processing 86, Elsevier/North Holland, 1986.

[10] P. Verissimo and H. Kopetz, "Design of distributed real-time systems," in Shape Mullender, editor, Distributed Systems, chapter 19. Addison-Wesley, 2nd edition, 1995.