

Fault Tolerance in Scalable Agent Support Systems: Integrating DARX in the AgentScape Framework

Benno Overeinder¹ Frances Brazier¹

¹ Department of Computer Science
Vrije Universiteit Amsterdam
de Boelelaan 1081a, 1081 HV Amsterdam
The Netherlands
{bjo, frances}@cs.vu.nl

Olivier Marin^{2,3}

² Labo. d'Informatique du Havre, University of Le Havre
25 rue Philippe Lebon, BP540 76058 Le Havre Cedex
³ Labo. d'Informatique de Paris 6, University Paris 6 – CNRS
4 place Jussieu, 75252 Paris Cedex 05, France
olivier.marin@univ-lehavre.fr

Abstract

Open multi-agent systems need to cope with the characteristics of the Internet, e.g., dynamic availability of computational resources, latency, and diversity of services. Large-scale multi-agent systems employed on wide-area distributed systems are susceptible to both hardware and software failures. This paper describes AgentScape, a multi-agent system support environment, DARX, a framework for providing fault tolerance in large scale agent systems, and a design for the integration of the two.

1. Introduction

Scalability in large scale multi-agent systems refers to both the number of agents and the number of hosts over which an application is distributed (and the “distance” between the hosts). Agents are independent, autonomous, mobile processes. Multi-agent systems are often, in fact, distributed cooperative applications on the Internet. As a result open multi-agent systems need to cope with the characteristics of the Internet, e.g., dynamic availability of computational resources, latency, and diversity of services. They are also susceptible to both hardware and software failures. Fault tolerance is required.

By ensuring the continuity of computations in spite of failure occurrences, fault-tolerant mechanisms are essential in large-scale environments.

It has been shown that replication is the most efficient reliability technique in the presence of failures [5]. However, software replication is a costly solution as it implies the multiplication of resource-consuming components as well as the consistency maintenance between replicas. It might be argued that increased resource consumption is not really a problem in a large-scale environment where resource

availability is virtually infinite. However, replicating every agent in an application comprising up to millions of agents is likely to undermine the overall determination of the result, particularly in terms of performance. The authors reckon that at any given time, some agents can be lost without significant incidence on the rest of the computation whereas some others are crucial to it. Moreover, this relative importance that every agent has within the application—referred to as its “criticality”—is expected to evolve dynamically.

Based on such reasoning, this paper intends to provide insight on how scalable support for agent-oriented applications can be designed, with dependability, and fault tolerance.

AgentScape [17] is a multi-agent support infrastructure. The middleware has been designed to be extensible, to support agents designed for different platforms, securely. Extensive mechanisms for fault tolerance have not, as yet, been implemented. DARX [10] is a framework aimed at building reliable software that would prove to be both flexible and scalable. This paper proposes to adapt the fault tolerance solutions developed for DARX in AgentScape. The paper is organized as follows. First, the importance of the dual aspect of agents is underlined: how they are defined in two close yet distinct research fields, namely Artificial Intelligence and Computer Systems. A description of AgentScape follows, with highlights on the way scaling is handled. Then, the main fault tolerance supporting techniques introduced in DARX are presented, as well as the way they are integrated in AgentScape. Finally, conclusions are outlined, and perspectives drawn.

2. Related work

Employment of intelligent agents on wide-area distributed networks incorporates both concepts from Artificial

Intelligence (AI) and methods and techniques from Computer Systems (CS). In this section some of the AI concepts basic to intelligent agents are discussed in relation to CS methods and techniques. Next, fault tolerance is described in relation to dependable multi-agent systems, bringing on the full complexity of distributed computer systems.

2.1. Agents: Artificial Intelligence and Computer Systems perspectives

From a Computer Systems perspective an agent is a process, a piece of running code with data and state. In Artificial Intelligence the functionality of these agents are most often described in terms of human behaviour, and to which the predicate intelligent is associated [18]. Agents are entities that are autonomous and pro-active (capable of making “their own” decisions when they like), have social ability (communicate with other agents), are reactive (can interact with objects and services), and may be mobile.

Agents interact with objects. Objects are passive [7]. In other words, an object needs to be invoked in order to perform a function, and performs only during an invocation. Agents, on the other hand, receive messages and autonomously decide if, when, and how to (re-)act. The only way for one agent to influence another agent is by sending a message, possibly with a request. An agent is free to ignore or react to such requests.

Agents can access services provided by others. Services may be either active or passive.

Agents in computer systems are often mobile. The ability of migration provides mobile agents a means to overcome the high latency or limited bandwidth problem of traditional client-server interactions by moving their computations to required resources or services. Migration also provides a means to protect data. Agents may need to migrate to given locations to view and process specific data.

A distinction can be made between migration in which the execution state is migrated along with the unit of computation or not [13]. Systems providing the former option are said to support *strong mobility*, as opposed to systems that discard the execution state across migration, and are hence said to provide *weak mobility*. In systems supporting strong mobility, migration is completely transparent to the migrated program, whereas with weak mobility, extra programming is required in order to save part of the execution state.

Strong mobility requires that the entire state of the agent, including its execution stack and program counter, is saved before the agent is migrated to its new location. Strong mobility is a complicated task to realize, and typical implementations of this functionality in multi-agent platforms provide platform specific solutions. As a consequence, interoperability between heterogeneous multi-agent systems

is difficult, if not impossible, to realize.

Many of the multi-agent platforms support weak mobility. Most of the agent systems are implemented on top of the Java Virtual Machine (JVM), which provides with object serialization basic mechanisms to implement weak mobility. The JVM does not provide mechanisms to deal with the execution state.

2.2. Fault-tolerant multi-agent systems

Research on fault tolerance in multi-agent systems mainly focuses on the ability to guarantee the continuity of every agent computation. This approach includes the resolution of consistency problems amongst agent replicas. However, some solutions also address the complex problems of maintaining agent cooperation [9], providing reliable migration for independent mobile agents and ensuring the exactly-once property of mobile agent executions [14].

Several solutions use specific entities to protect the computational elements of multi-agent systems [6, 8, 9]. These approaches specify the control of agents separately from the functionalities of the multi-agent system, which can be used to improve fault tolerance.

In [6], sentinels represent the control structure of the multi-agent system. Each sentinel is specific to a functionality, handles the different agents which interact to provide the corresponding service, and monitors communications in order to react to agent failures. Adding sentinels to a multi-agent system seems to be a good approach, however the sentinels themselves represent bottlenecks as well as failure points for the system.

A similar architecture is that of the Chameleon project [8]. Chameleon is an adaptive fault tolerance system using reliable mobile agents. The methods and techniques are embodied in a set of specialized agents supported by a fault tolerance manager (FTM) and host daemons for handshaking with the FTM via the agents. Adaptive fault tolerance refers to the ability to adapt dynamically to the evolving fault tolerance requirements of an application. This is achieved by making the Chameleon infrastructure reconfigurable. Static reconfiguration guarantees that the components can be reused for assembling different fault tolerance strategies. Dynamic reconfiguration allows component functionalities to be extended or modified at runtime by changing component composition, and components to be added to or removed from the system without taking down other active components. Unfortunately, through its centralized FTM, this architecture suffers from the same problems as the previous approach.

Kumar *et al.* [9] present a fault tolerant multi-agent architecture that regroups agents and brokers. Similarly to [6], the agents represent the functionality of the multi-agent system and the brokers maintain links between the agents.

Kumar *et al.* [9] propose to organize the brokers in hierarchical teams and to allow them to exchange information and assist each other in maintaining the communications between agents. The brokerage layer thus appears to be both fault-tolerant and scalable. However, the implied overhead is tremendous and increases with the size of the system. Besides, this approach does not address the recovery of basic agent failures.

To solve the overhead problem, Fedoruk and Deters [4] propose the use of proxies. This approach tries to make the use of agent replication transparent; that is, computational entities are all represented in the same way, disregarding whether they are a single application agent or a group of replicas. The role of a proxy is to act as an interface between the replicas in a replicate group and the rest of the multi-agent system. It handles the control of the execution and manages the state of the replicas. To do so, all the external and internal communications of the group are redirected to the proxy. A proxy failure isn't crippling for the application as long as the replicas are still present: a new proxy can be generated. However, if the problem of the single point of failure is solved, this solution still positions the proxy as a bottleneck in case replication is used with a view to increasing the availability of agents. To address this problem, the authors propose to build a hierarchy of proxies for each group of replicas. They also point out the specific problems which remain to be addressed: read/write consistency and resource locking, which are discussed in [15] as well.

3. AgentScape: A scalable multi-agent infrastructure

AgentScape is a middleware layer that supports large-scale agent systems. The rationale behind the design decisions are (i) to provide a platform for large-scale agent systems, (ii) support multiple code bases and operating systems, and (iii) interoperability with other agent platforms [17].

3.1. The AgentScape model

The overall design philosophy is “less is more,” that is, the AgentScape middleware should provide a minimal but sufficient support for agent applications, and “one size does not fit all,” that is, the middleware should be adaptive or reconfigurable such that it can be tailored to a specific application (class) or operating system/hardware platform.

Agents and *objects* are basic entities in AgentScape. A *location* is a “place” at which agents and objects can reside (see Fig. 1). Agents are active entities in AgentScape that interact with each other by message-passing communication. Furthermore, agent migration in the form of weak mobility is supported. Objects are passive entities that are only

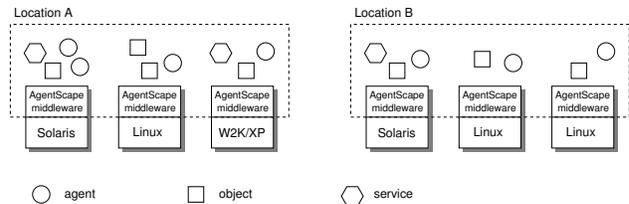


Figure 1. AgentScape conceptual model.

engaged into computations reactively on an agent's initiative. Besides agents, objects, and locations, the AgentScape model also defines *services*. Services provide information or activities on behalf of agents or the AgentScape middleware.

Scalability, heterogeneity, and interoperability are important principles underlying the design of AgentScape [12]. For example, scalability of agents and objects is realized by distributing objects according to a per-object distribution strategy, but not agents. Instead, agents have a public representation that may be distributed if necessary. Agent-agent interaction is exclusively via message-passing communication. Asynchronous message-passing has good scalability characteristics with a minimum of synchronization between the agents.

3.2. AgentScape architecture

The *AgentScape Operating System* (AOS) forms the basic fundament of the AgentScape middleware. An overview of the AgentScape architecture is shown in Fig. 2. The AOS offers a uniform and transparent interface to the underlying resources and hides various aspects of network environments, communication, operating system, access to resources, etc. The AgentScape API is the interface to the middleware. Both agents and services (e.g., resource management and directory services) use this interface to the AOS middleware.

Agents and objects are supported by *agent servers* and *object servers* respectively. Agent servers provide the interface and access to AgentScape to the agents that are hosted by the agent server. Similarly, objects servers provide access to the objects that are hosted by the object server. Services are made accessible in AgentScape by service access providers.

A location is a closely related collection of agent and object servers, possibly on the same (high-speed) network, on hosts which are managed in the same administrative domain. Each host runs a *minimal* AOS kernel, and zero or more agent servers, objects servers, and service access providers. A location is implemented by the distributed AOS kernels, the agent servers, the object servers, and service access providers.

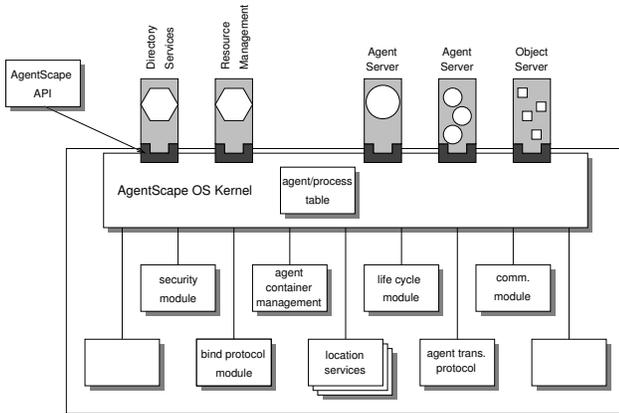


Figure 2. An AgentScape middleware architecture.

Depending on the policy or resource requirements, one agent can be exclusively assigned to one agent server, or a pool of agents can be hosted by one agent server. The explicit use of agent servers makes some aspects in the life cycle model of agents more clear. An active agent is assigned to, and runs on a server; a suspended agent is not assigned to an agent server. In this model, starting a newly created, or activating an existing suspended agent, is similar, and some design decisions of the agent life cycle can be simplified [2].

The design of the AgentScape Operating System is *modular*. The AOS kernel is the central active entity that coordinates all activities in the middleware. The modules in the AOS middleware provide the basic functionality. Below a brief overview of the most important modules is given. The life-cycle module is responsible for the creation and deletion of agents. The communication module implements a number of communication services, e.g., similar to TCP, HTTP, and streaming, with different qualities-of-service. Support for agent mobility is implemented in the agent transfer protocol module. The location service associates an agent identifier with a contact address. There are also location services for objects, services, and locations. The security architecture is essential in the design of AgentScape, as it is an integral part of the middleware. Many modules in the middleware have to request authentication or authorization in order to execute their tasks.

3.3. AgentScape prototype

A prototype implementation of the AgentScape architecture is currently available and provides the following basic functionality: creation and deletion of agents, communication between agents and middleware, and weak migration of agents. The AgentScape Operating System kernel and

some basic services are implemented in the programming language Python, while the agent servers are implemented in Java and Python. Agent servers for other programming languages will be made available in forthcoming releases of AgentScape.

Distributed shared (replicated) objects in AgentScape will be supported by the Globe system [16]. Globe is a large-scale wide-area distributed system that provides a object-based framework for developing applications.

The use of multiple programming languages is not only available at the application level (i.e., building agents and objects in a programming language of choice), but also the modules of the AOS are implemented in different programming languages. For example, multiple location services can be present in the AOS, each implemented in a different language.

4. DARX-inspired fault tolerance mechanisms

DARX is a framework designed to support the development of fault-tolerant applications built upon multi-agent systems. Although DARX is implemented and works as stand-alone middleware, most of its underlying concepts may be used independently in other platforms. This section details the concepts that are to be integrated in AgentScape.

Throughout this section, a distributed system is assumed, in which agents are independent processes; they communicate by message-passing only. Processes are assumed to be fail-silent. Once a specific process is considered as having crashed, it cannot participate to the global computation anymore. Byzantine behaviours might be resolved, but are not yet integrated in the failure model.

4.1. Fundamental notions

The following notions constitute the abstract foundations of the DARX framework. Adapted to AgentScape, they will bring the functionalities required for ensuring basic fault tolerance.

4.1.1. Replication management

DARX provides fault tolerance through software replication. It is designed in order to be agent-dependent, as opposed to location-dependent¹. That is, if a machine/location crashes, it can be restarted but no server information needs to be recovered. However, the state of the agent replicas which were present on the location may be recovered. The

¹A location is an abstraction of a physical location. It hosts resources and processes, and possesses its own unique identifier. DARX uses a URL and a port number to identify each location.

reason for this choice comes from the fundamental assumption that the *criticality* of an agent² evolves over time; therefore, at any given moment of the computation, all agents do not require to benefit from the same fault tolerant mechanisms, if any at all. Hence on every location, some agent replicas will be kept consistent with pessimistic strategies, others with optimistic ones, while some others will not be replicated at all.

In order to minimize interference with the application development process, replication management is kept *transparent* to the supported application. While the latter deals with agents, DARX handles replication groups. Each of these groups consists in software entities (replicas) which represent the same agent. Thus in the event of failures, if at least one replica is still up, then the corresponding agent is not lost to the application.

To sum up, agent-dependent fault tolerance is enabled by the notion of replication group (RG): the set of all the replicas which correspond to a same agent. It follows that a RG contains at least one active replica. Within the RG, replicas must be kept consistent so as to ensure recovery in case of failures. To allow for this, several replication strategies are made available by the DARX framework. The strategies offered can be classified in two main types: (1) *active*, where all replicas process the input messages concurrently, and (2) *passive*, in which only one replica is in charge of the computation while periodically transmitting its state to the other replicas.

One of the innovative aspects of DARX is that several strategies may coexist inside the same RG. As long as one of the replicas is active, meaning that it executes the associated agent code and participates to the application communications, there is no restriction on the activity of the other replicas. Indistinctly, those other replicas may be backups (passive strategy) or followers (semi-active strategy) of the active replica, or even equally active replicas. Furthermore, it is possible to switch from a strategy to another with respect to a replica: a follower may become a backup, and so on ...

Hence a considerable amount of information is necessary to describe a replication group:

- the *criticality* of its associated agent,
- its replication degree—the number of replicas it contains,
- the list of these replicas, ordered by potential of leadership³,

²The *criticality* of a process defines its importance with respect to the rest of the application. Obviously, its value is subjective and evolves over time. For example, towards the end of a distributed computation, a single agent in charge of federating the results should have a very high *criticality*; whereas at the application launch, the *criticality* of that same agent may have a much lower value.

³The potential of leadership is the capacity of a replica to represent its

- the list of the replication strategies applied inside the group,
- the mapping between replicas and strategies.

The sum of these pieces of information constitutes the replication policy of a RG. A replication policy must be reevaluated in three cases: (1) when a failure inside the RG occurs, (2) when the *criticality* value of the associated agent changes, and (3) when the policy cannot be enforced due to environment variations such as CPU or network overloads. It seems obvious that all the replicas of a group must have a consistent and up-to-date version of their policy. However, since the replication policy may be reassessed frequently, it appears reasonable to centralize this decision process. A leader is elected within the RG for this purpose. Its objective is to adapt the replication policy to the *criticality* of the associated agent as a function of the characteristics of its context—the information obtained through observation. As mentioned earlier, DARX allows for dynamic modifications of the replication policy. Replicas and strategies can be added to or removed from a group during the course of the computation, and it is possible to switch from a strategy to another on the fly. For example, the recovery of a missing active replica may be decided by activating the most suitable backup inside the RG; or if a backup crashes, a new replication can be initiated to maintain the level of reliability within the group; or if the *criticality* of the associated agent decreases, it is possible either to suppress a replica or to switch the strategy attached to a replica from an active form to a passive one.

Figure 3 depicts the composition of a replica. In order to benefit from fault tolerance abilities, each agent gets to inherit the functionalities of a `DarxTask` object, enabling DARX to control the agent execution. Each task is itself wrapped into a `TaskShell`, which handles the agent inputs/outputs. Hence DARX can act as an intermediary for the agent, committed to deciding when an agent replica should really be started, stopped, suspended or resumed, and exactly when and which message receptions should take effect. Leaders are wrapped in enhanced shells, comprising an additional `ReplicationManager`. This manager collects information about the environment—network load, memory available, etc.—and performs the periodical reassessment of the replication policy. It also maintains the group consistency by sending the relevant information to the other replicas, following the policy requirements.

Communication between agents passes through proxies implemented by the `RemoteTask` interface. These prox-

RG. The strategy used to keep a replica consistent is the main parameter in the calculation of this potential; the more pessimistic the strategy, the higher its potential. The other parameters emanate from the DARX-integrated observation service; they include the load of the host, the date of creation of the replica, the latency in the communications with the other replicas of the group, ...

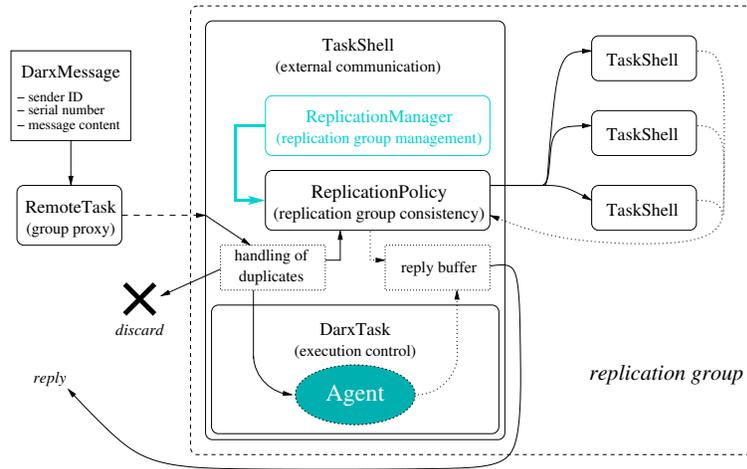


Figure 3. Replication management scheme

ies reference replication groups; it is the naming service which keeps track of every replica to be referenced, and provides the corresponding `RemoteTask`.

Figure 4 shows a tiny agent application as seen in the DARX context. An emitter, agent B, sends messages to be processed by a receiver, agent A. At the moment of the represented snapshot, the value of the *criticity* of agent B is minimal; therefore the RG which represents it contains a single active replica only. The momentary value of the *criticity* of agent A, however, is higher. The corresponding RG comprises three replicas: (1) an active replica A elected as the leader, (2) a follower A' to which incoming messages are forwarded, and (3) a backup A'' which receives periodical state updates from A.

In order to transmit messages to A, B requested the relevant `RemoteTask` RTA from the naming service. Since replication group A contains only one active replica, RTA references replica A and no other.

4.1.2. Failure detection

DARX includes a failure detection service based on an adaptable implementation of the unreliable failure detector [1].

The failure detection serves a major goal: to maintain dynamic lists of the valid replicas participating to the application. Failure detectors are organized in groups; they exchange heartbeats and maintain a list of the processes which are suspected of having crashed. Therefore, in an asynchronous context, failures can be recovered more efficiently. For instance, the failure of a process can be detected before the impossibility to establish contact arises within the course of the supported computation—process A may be suspected of having crashed before a process B fails to

contact it for application purposes.

Every DARX entity integrates an independent thread which acts as a failure detector. It sends heartbeats to the other group members and collects the incoming heartbeats. Once an entity has been suspected several times in a row, it is considered as having crashed. A detector which suspects a crash submits the information to the rest of the group. If all the other group members confirm the crash, then the suspected entity is excluded from the group. Obviously it is possible that a group leader will be excluded; therefore such an event triggers a new election.

Failure detection may also be used to convey internal communications. Information can be exchanged between entities via piggybacking on the failure detection heartbeats.

4.1.3. Replication policy adjustment

DARX aims at providing decision-making support so as to fine-tune the fault tolerance for each agent with respect to its evolution and that of its context. Decisions of this type may only be reached through a fairly good knowledge of the dynamic characteristics of the application and of the environment. Analysis of the original agent source code might provide enough required information to enable DARX support without further modifications of the original program [3]. However, at this point of the research project, the application developer must respect a few guidelines and constraints which are given thereafter. As a Java framework, DARX includes several generic classes which assist the developer through the process of implementing a reliable multi-agent application. The choice of those generic classes comes from the study of the OMG MASIF [11] specifications, as well as that of the most recurrent aspects of various MAS, therefore DARX-compliant application building is very close to most

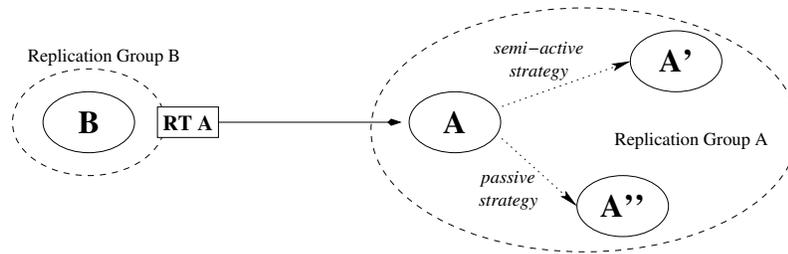


Figure 4. A simple agent application example

agent developing environments.

Every agent class must extend a `DarxTask` for several reasons. First, because the `DarxTask` is the point where DARX handles the execution of an agent: application-specific control methods to start, stop, suspend and resume the agent have to be defined for this purpose. Such methods would be very hard to implement in a general context, where the application developer would not have to intervene, without modifying the JVM and the efficiency loss would be drastic. Also, the state of an agent is essential in determining its criticality. Any number of states can be defined for the agent; each of these states is to be mapped to a corresponding *criticality* in the code of the `ReplicationManager`, and to every criticality corresponds a user-defined replication policy which will be applied at runtime.

Finally, the application designer can conceive an unlimited number of replication policies. Although generic replication strategies are implemented, fault tolerance protocols that are specific to the application can be developed. DARX provides a generic `ReplicationStrategy` class which may be extended to fulfill the needs of the programmer. Basic methods allow to define the consistency information within the group, as well as the way this information ought to be propagated in different cases, such as synchronous or asynchronous messages for example. A few simple strategies are already built in DARX; others are undergoing research, like quorum-based strategies for instance.

4.2. Integration in AgentScape

AgentScape provides a means for large-scale deployment and interoperability between platforms. Instead of trying to interconnect both the DARX and AgentScape middleware systems while running them concurrently, the current design is to integrate DARX components into AgentScape. The modular architecture of both DARX and AgentScape makes this possible.

Agents developed along the DARX guidelines (see Section 4.1.3), make use of a DARX specific runtime system (RTS) and are hosted by regular AgentScape agent servers. Effectively, agents are extended with the DARX RTS to pro-

vide functionality for fault tolerance⁴ (see Fig. 3). Such agents may call the DARX RTS at any point in their life cycle, thus initiating DARX-enabling processes at the agent level: failure detection and replication management. The DARX fault tolerant RTS associated with the agent can make calls to the AgentScape middleware to communicate with the peers in the replication group or to obtain monitoring information for the observation service, etc. Monitoring information (e.g. load of host, date of creation of replicas, communication latencies to other replicas in the group), is provided by an observation service that acquires its information from the AgentScape management system.

Replication management does not require modification of the original DARX implementation, as DARX agents are designed perform replication management within each replication group. The policy for each group is known to each replica.

An extension of the current model is to adapt DARX's failure detection mechanism for replication groups. Instead of servers being responsible for checking the liveness of their neighbours as well as their hosted agents, all replicas will work at detecting failure occurrences inside their own groups. This is still an area of research. As more than one failure detection relationship may exist between two hosts, the the cost of modification may be significant. This needs to be further explored.

5. Summary and future work

A first design is presented of the integration of DARX agent replication to support fault tolerance in the AgentScape framework. The design premises of both DARX and AgentScape are that the software systems must be open, i.e., able to be used in combination with other software systems. With the integration of both systems, the added value of these design requirements greatly simplified the integration effort.

The management system [2] for the next prototype of AgentScape will incorporate the necessary monitor func-

⁴DARX agents make use of `TaskShell` to acquire the fault tolerance functionality.

tionality required for the observation service of the DARX framework. The information used by the observation service (i.e., load of host, date of creation of replicas, communication latencies to other replicas in the group, etc.) are monitored by the AgentScape management system. By defining new monitor filters, more system information can be obtained for the management system, if necessary.

Future developments will also include the implementation of DARX specific agent servers, making DARX fault tolerant functionality (including replication management and failure detection) the responsibility of the AgentScape middleware. Individual agents will no longer need to be extended with a DARX RTS, and multiple failure detection relationships between hosts will no longer be necessary.

Besides fault tolerance, scalability and performance are important issues of the AgentScape/DARX system. Performance evaluation through experimentation is planned to assess the scalability and performance of the fault tolerant agent support system.

Acknowledgements

This research is supported by the NLnet Foundation, <http://www.nlnet.nl/>. The authors would like to thank Etienne Posthumus and Maarten van Steen from the Vrije Universiteit Amsterdam, and Jean-Pierre Briot, Pierre Sens, and Zahia Guessoum from LIP6 for their valuable contributions.

References

- [1] M. Bertier, O. Marin, and P. Sens. Implementation and performance evaluation of an adaptable failure detector. In *Proceedings of the International Conference on Dependable Systems and Networks*, Washington, DC, June 2002.
- [2] F. M. T. Brazier, D. G. A. Mobach, B. J. Overeinder, and N. J. E. Wijngaards. Managing agent life cycles in open distributed systems. In *Proceedings of the ACM Symposium on Applied Computing (SAC 2003)*, Melbourne, FL, Mar. 2003.
- [3] J.-P. Briot, Z. Guessoum, S. Charpentier, S. Aknine, O. Marin, and P. Sens. Dynamic adaptation of replication strategies for reliable agents. In *Proceedings of the 2nd Symposium on Adaptive Agents and Multi-Agent Systems (AAMAS-2)*, London, UK, Apr. 2002.
- [4] A. Fedoruk and R. Deters. Improving fault-tolerance by replicating agents. In *Proceedings of the 1st International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, Bologna, Italy, July 2002.
- [5] R. Guerraoui and A. Schiper. Software-based replication for fault tolerance. *IEEE Computer*, 30(4):68–74, Apr. 1997.
- [6] S. Hägg. A sentinel approach to fault handling in multi-agent systems. In *Proceedings of the 2nd Australian Workshop on Distributed AI, 4th Pacific Rim International Conference on Artificial Intelligence (PRICAI'96)*, Cairns, Australia, Aug. 1996.
- [7] N. R. Jennings and W. J. Wooldridge, editors. *Agent Technology: Foundations, Application, and Markets*. Springer-Verlag, Berlin, Germany, 1998.
- [8] Z. Kalbarczyk, R. K. Iyer, S. Bagchi, and K. Whisnant. Chameleon: A software infrastructure for adaptive fault tolerance. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):560–579, June 1999.
- [9] S. Kumar, P. R. Cohen, and H. J. Levesque. The adaptive agent architecture: Achieving fault-tolerance using persistent broker teams. In *Proceedings of the 4th International Conference on Multi-Agent Systems (ICMAS 2000)*, Boston, MA, July 2000.
- [10] O. Marin, P. Sens, J.-P. Briot, and Z. Guessoum. Towards adaptive fault-tolerance for distributed multi-agent systems. In *Proceedings of the European Research Seminar on Advances in Distributed Systems (ERSADS'2001)*, pages 195–201, Bertinoro, Italy, May 2001.
- [11] D. Milojevic et al. MASIF: The OMG mobile agent system interoperability facility. In *Proceedings of the 2nd International Workshop on Mobile Agents (MA'98)*, volume 1477 of *LNCS*, pages 50–67, Stuttgart, Germany, Sept. 1998.
- [12] B. J. Overeinder, E. Posthumus, and F. M. T. Brazier. Integrating peer-to-peer networking and computing in the AgentScape framework. In *Proceedings of the 2nd IEEE International Conference on Peer-to-Peer Computing*, pages 96–103, Linköping, Sweden, Sept. 2002.
- [13] G. P. Picco. Mobile agents: An introduction. *Microprocessors and Microsystems*, 25(2):65–74, Apr. 2001.
- [14] S. Pleisch and A. Schiper. Fatomas—A fault-tolerant mobile agent system based on the agent-dependent approach. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks*, pages 215–224, Göteborg, Sweden, July 2001.
- [15] L. Silva, V. Batista, and J. Silva. Fault-tolerant execution of mobile agents. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 135–143, New York, NY, June 2000.
- [16] M. van Steen, P. Homburg, and A. S. Tanenbaum. Globe: A wide-area distributed system. *IEEE Concurrency*, 7(1):70–78, Jan.–Mar. 1999.
- [17] N. J. E. Wijngaards, B. J. Overeinder, M. van Steen, and F. M. T. Brazier. Supporting Internet-scale multi-agent systems. *Data and Knowledge Engineering*, 41(2–3):229–245, June 2002.
- [18] M. J. Wooldridge and N. R. Jennings. Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, June 1995.