Achieving dependability in a system consisting of autonomous agents

Ralph Deters
Department of Computer Science
University of Saskatchewan,
Saskatoon, Canada

ralph@cs.usask.ca

Abstract
The runtime behavior of complex systems consisting of large numbers of autonomous, heterogeneous and highly connected agents is difficult to predict. The lack of information concerning the state of the individual agents and their interconnections makes it hard to *understand* the current system state and to decide if and how to *react*. This paper investigates some issues concerning dependability of autonomous agent-based systems.

Motivation

When developing a system consisting of several agents control becomes an important issue. As long as the system is small and homogeneous, it is possible to control it and to take *all* decisions in a centralized way. However, if the system is consisting of a large number of heterogeneous and highly interconnected agents, the centralization of control comes at the price of extreme complexity and ultimately the system may become unmanageable.

The obvious solution is to decentralize control by introducing agents responsible for controlling particular parts of the system. Centralized control and decentralized control is a continuum, which starts with a single component responsible for controlling everything and ends with every agent controlling itself.

The obvious advantage of a more decentralized control via granting autonomy to the agents is that low-level control decisions are taken locally. Autonomous adapt to changes themselves without the need of central monitoring. And as long as the conflicts and problems are locally solvable, a system consisting largely of autonomous agents shows a satisfying behavior.

However, in case of unexpected events that are not locally detectable and/or treatable, such a local control may become no longer possible. The autonomous agents designed to achieve a local optimum behave now in a very unpredictable way making it hard to reach a stable system state.

The obvious solution of shifting the control upwards doesn't really help since it would result to the very undesirable effect that in a critical situation the centralized control components are overloaded with local decisions limiting their ability to diagnose and treat the situation.

What is dependability?
Dependability is a notion that hasn't been precisely defined in literature, but is typically associated with availability, reliability, security and safety. Since these terms are typically defined having general software components in mind [2], it is necessary to adapt the exiting definitions of these terms to agents.

Availability of an agent
Agent availability is defined as the expected fraction of time during which an agent is responding in a meaningful way to messages, perceived changes in the environment and changes regarding its own state. Availability is computed by calculating the ratio of up time to the sum of up time plus down time. (Down time is the product of failure intensity and the mean time to repair/recovery.)

Reliability of an agent
The reliability of an agent expresses the probability of failure-free operation for a specified time in a specified environment. Reliability expresses the ability of the agent to cope with faults in a way that no errors or failures appear. We measure reliability by observing if the agent is able to work in an average situation over a specified period of time.

Security of an agent
We consider an agent secure, if it takes protective measures that ensure a state of inviolability from hostile acts or influences. Security expresses the ability of the agent to cope with unexpected events like message bursts, denial of a resource and failures of other agents.

Safety of an agent
We define safety as the absence of risk. An agent can be considered safe if its activities do not endanger other agents. Safety is measured as the reciprocal of the probability that the actions of the agent lead to a failure of another agent.

Measuring of the availability, reliability, security and safety may vary from agent to agent. Therefore it is important to define in form of a specification for every agent degrees of reliability, availability, security and safety, which the agent is expected to show and ways to measure them. If this information is available, it is possible to evaluate the agents' internal state by constant monitoring.
It is particularly interesting to let the agent perform this task itself in order to ensure its autonomy. It is feasible to built a self-controlling agent, which is able to observe its own behavior, is able to reason about its internal state and to perform simple actions to preserve/ reach desirable states.

An example
Agents are more or less autonomous objects able to communicate beyond normal method invocation, just by sending messages to each other. Their ability to communicate via a standardized message interface using a common protocol enables the easy development

of open systems, which can be changed by adding, modifying or removing agents. Unfortunately, this flexibility of agents comes at the price of increased system complexity and ultimately a more chaotic/unpredictable behavior of the system
In absence of information concerning the states of the agents, their functional dependencies and resource usage/requirements makes it extremely difficult to determine the current state and to decide if and how to intervene to guarantee a desired system state.
Figure 1 shows a distributed system consisting of two processes each hosting three agents. If no information about the agent's states is available, we are unable to determine the system-state.

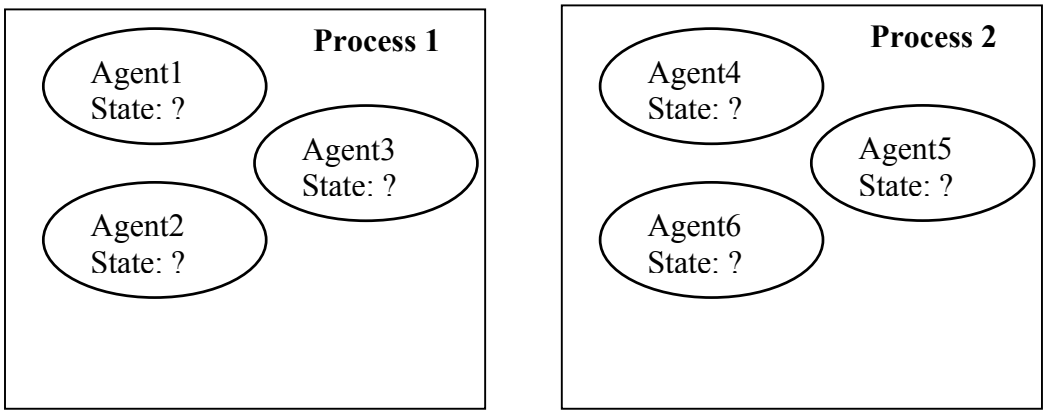

Figure 1: Unknown agent states

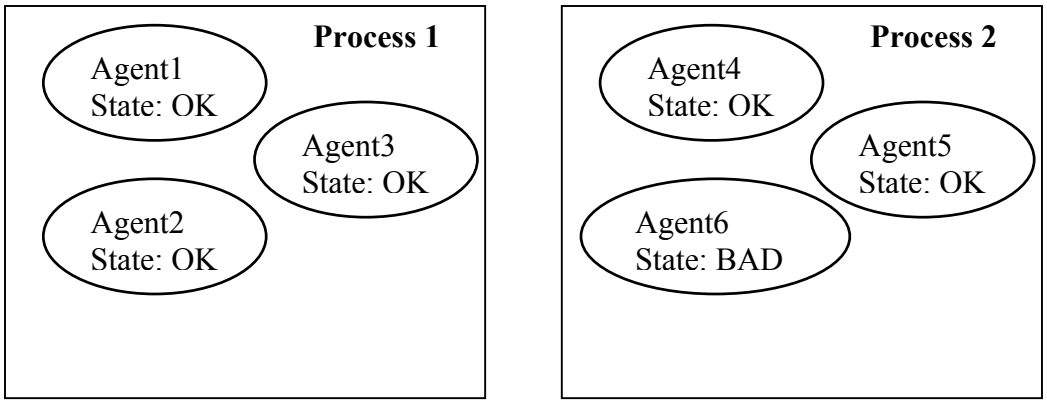

Figure 2: Known agent states

If the state information becomes available (Figure 2), it is possible to see that one agent has problems. But the impact on the other agents is impossible to predict without the information about the functional dependencies among the agents.

Let's assume that the following functional dependencies for the agents are known:

```
Agent1 depends on Agent2
Agent2 depends on Agent6
Agent5 depends on Agent4
Agent4 depends on Agent1
Agent3 depends on Agent2
```

Suddenly the importance of the Agent6 becomes clear and the system-state can be assumed as highly critical.

By adding the information about the resource requirements/usage of the agents, it is possible to determine the cause of the problem. If in addition possibilities exist to gain control over the resources e.g. withdraw/redistribute we are able to move from a critical[1] to a non-critical situation.

The MIB

To enable easy access of the states and functional dependencies the information should be stored in a way that it is accessible in an easy way for other components. Instead of obtaining this information by request from the agents it seems more reasonable to introduce a central management information base (MIB), which is able to store all information relevant to system management.
To model the existing agent-based system in the MIB the concept of "managed object" (MO) is used. A MO is the smallest entity in the model. N:M relations exit between the agents and the MOs allowing that a single agent is represented by one or more MOs and that a MO may represent multiple agents. In addition, a MO can consist of other MOs allowing the introduction of abstraction levels.
Whenever a change of an attribute in one MO appears, a consistency check is performed to ensure the consistency of the abstract MOs. In this way a simple propagation is possible which enables us to reduce the effort of monitoring.

The MIB is used to store

- the name and location of every agent
- the state of every agent
- the resource usage of every agent
- the resource demands for every agent
- the known functional dependencies
- the state of every agent

While it is in principle possible to distribute the MIB in the current implementation, a centralized non-distributed MIB is used.
The MIB is only useful if the data contained in it is up to date. The problem is how to obtain the information. Three basic approaches are possible

[1] Critical is a state in which the dependability constraints are violated

By request - Pull
In this approach whenever the information is needed it is requested from the agent by sending a request message.

By report - Push
Here the agent is supposed to report this information. This can be done by sending an initial report and in case of any change sending an update.

By observation - Monitoring
The last and most arduous approach would be monitoring the agent's behavior e.g. messages received and sent.

To minimize the message traffic the push approach in which the agent reports without further notice is most favorable. The messages should be send to special entities called event streams. Every event stream has several agents attached to it (event producers) and at least one observer (event consumer).
Several event streams are possible and in order to distribute the load this is most desirable. The task of the observer is to analyze the incoming messages and to produce update commands for the MIB.

Message types
The messages sent to the event stream have to be easy to evaluate. In the current version of the system we decided that the messages are

Error Messages
An error message consists of the sender id, error code, time stamp, and the assumed cause that lead to the error. Error messages are used to inform that the agent has reached a critical state.

Warning Messages
Warning messages consist of the sender id, warning code, time stamp and the assumed cause that lead to the error. Warning messages are used to inform that the agent is about to reach a critical state.

Clearance Messages
A clearance message consists of a reference to a previous warning or error message and a time stamp. The agent signals with this message that a cause for a previous error or warning message has been removed.

State-change Messages
A state-change message consists of the sender id, the previous state, the new state and a time stamp indicating the moment of the state change.

Service Message
This message consists of a sender id, and a list of services that are currently offered by the agent.

Dependency report Message
This message consists of a list of agents, services or resources that are vital for this agent.

Observation Message
An observation message consists of the sender id and a nested message of one of the above mentioned types.

Agents observing other agents
Relying only on the push of the messages from the agents can be dangerous since in the most critical states an agent is often not able to send a message. This would leave the MIB unaware of the critical states of agents.
Therefore we enable the agents to observe their environment and to report about changes. This means that agents report about their own states, resource usage etc. and about anomalies e.g. unexpected behavior of other agents. In this way a self-controlling society of agents is possible. Another advantage of allowing agents to report about another agents is that a combination of transparent and less/ non-transparent agents is possible. This is of high importance for us since we are using already developed agents (non-transparent).
The problem arising with observation messages is that contradicting messages can appear in a message stream. It is possible that an agent1 sends a clearance message and that the agent2 sends an observation messages reporting about a critical state.

Analyzing the stream
The observer is more than a simple message driven component. Instead of observing individual messages it is necessary to correlate the messages [3]. The correlation of the messages is done by defining simple rule-like cases, for example:

```
IF
      (agent1.state == state s1 AND agent2.state == s2)
THEN
      UPDATE (agent3.state = "CRITICAL")
```

By defining a set of these cases it is possible to efficiently analyze the stream of event messages [4]. To ensure that the cases can apply also to new situations, a simple adaptation procedure is used which enables the runtime modification of the cases. In this way a simple case-based reasoning is performed which is fast enough to handle the flood of event messages and flexible enough to adapt to new situations.

Potential threats
To ensure the desired dependability of the system it is useful to analyze the potential threats such a system could encounter. In the following a by no means complete list of

scenarios is given which will be used in the next section to develop general design criteria.

External events
Due to external events the environment of the agents can change. An example of such an event could be a drop in the available processor power. Changes in the environment can lead to the unannounced withdrawal of resources and the malfunction of other agents.

Internal events
Due to internal faults of the agent code it is possible that it fails to offer its services or that its doesn't work error free. In the worst case the failure of a single agent could lead to a chain reaction causing a reduced functionality of the whole system.

Resource conflicts
It is important to note that in every decentralized system, which has limited resources, conflicts are inherent. A typical resource conflict would arise if one agent uses extensively a resource, which therefore leads to a denial if requested by another agent.

Design goals
To ensure dependability it is important to enable the fast and reliable detection and treatment of critical states. We therefore propose the following design goals

Transparency of agents
As stated earlier it is important that the state, functional dependencies and the resource requirements of every agent are transparent. This can be achieved by storing this information in a management information base (MIB). Instead of obtaining this information by request from the agents it seems more reasonable to make them report in case of change to the MIB.

Self-awareness of agents
An agent should be aware of its actions, resource usage and functional dependencies. In order to be able to reflect and modify its own behavior it is necessary that agents consist of several threads.

Dependability of agents
Closely linked to the self-awareness of an agent is the constant monitoring of its reliability, availability, security and safety parameters. The agent should constantly compare the required parameters with the observed ones and change its behavior accordingly.

Observing the environment
An agent should also constantly monitor the environment to detect if any anomalies concerning other agents appear. Using agents to observe each other helps in the

early detection of a critical state in a single agent that might prevent it from reporting. Every assumed problem should be signaled immediately to the MIB.

Fault-tolerant
The agent should be able to cope with local faults without the need for external support.

Local treatment
If an agent observes a problem that it could treat itself it should treat the problem itself.

Obedience
The last design criteria we assume to be important is the ability of the agent to change its behavior according to instructions from higher control units. This again requires the ability of the agent to be aware of its behavior and to interrupt if necessary its current tasks.

Conclusion

This paper defines a notion of dependability for multi-agent systems. It describes the problems concerning the conflict between dependability and autonomy of agents. We propose a set of design goals (requirements) for achieving dependable autonomous agents. These requirements are underlying the design of a multi-agent based architecture for a peer help system in a university environment which is being developed at the University of Saskatchewan [1].

References

1. Greer, J., McCalla, G., Cook, J., Collins, J., Kumar, V., Bishop, A. and Vassileva, J. (1998) The Intelligent HelpDesk: Supporting Peer Help in a University Course, Proceedings ITS'98, San Antonio, Texas.
2. Musa J., Iannino, A., Okumoto, K. (1987) Software Reliability: Measurement, Prediction, Application. McGraw Hill: New York.
3. Deters R. (1994) Case-based Event Correlation. In Proceedings AI'94, Fourteenth International Avignon Conference, Paris, May 30 - June 3, 1994, pp. 323-331.
4. Deters R. (1995) Case-based Diagnosis of Multiple Faults, in Veloso, M. Aamodt A. (Eds.) Case-Based Reasoning Research and Development, Lecture Notes in Artificial Intelligence 1010, Springer Verlag: Berlin. pp. 411-420.