

Agent Dependability as an Architectural Issue

Peter C. Lockemann
lockemann@ipd.uka.de

Jens Nimis
nimis@ipd.uka.de

Fakultät für Informatik, Universität Karlsruhe (TH), Postfach 6980, D-76128 Karlsruhe, GERMANY

ABSTRACT

The paper demonstrates how a layered reference architecture for software agents provides a framework for a dependability model that allows to associate the sources of failures and the ensuing error handling with a specific layer, thus integrating dependability directly into the design of agents.

Categories and Subject Descriptors

C.4 [Performance of Systems]: Fault Tolerance; Reliability, Availability, and Serviceability; I.2.11 [Artificial Intelligence]: Distributed Artificial Intelligence—*multiagent systems, intelligent agents*; D.2.11 [Software Engineering]: Software Architectures

General Terms

Design, Reliability

Keywords

Multiagent System, Layered Software Architecture, Dependability

1. INTRODUCTION

Ideally, agents never fail. Practically, though, disturbances may occur that are outside the control of the agent system. We call a multiagent system dependable if it or its constituent agents include internal mechanisms that guarantee that all services are rendered according to specification even in the presence of disturbances.

The main thesis of this short paper is that if the design of agents and multiagent systems follows the principle of levels of abstraction resulting in a layered architecture, the principle will also determine the design of dependability by associating a specific failure and its remedies with an individual layer. In this way one can develop a well-structured dependability model where each kind of error handling can be isolated and localized.

2. REFERENCE ARCHITECTURE

Let us first introduce a layered reference architecture as the framework within which to integrate dependability in a systematic fashion and thus prove our thesis. The architecture as shown in Fig. 1

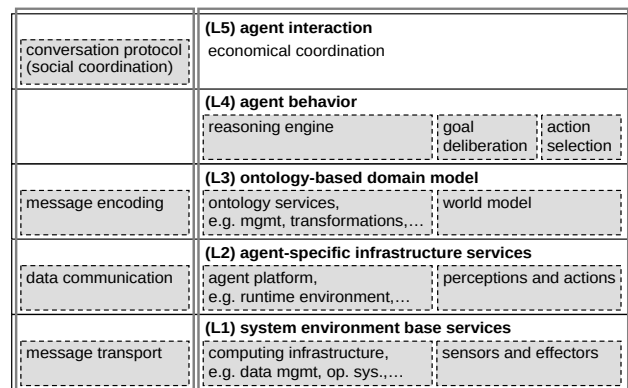


Figure 1: Reference architecture for a multi-agent system

was developed in [2] where it is explained in detail. The layers are based on the properties of intelligent software agents as listed by Wooldridge [7]. The lower two layers provide the basis for situatedness, autonomy, reactivity and proactivity, layer L3 for utility and goal-orientation. Layer L4 realizes agent behavior encompassing utility, reactivity and goal deliberation, while layer L5 adds sociability.

Sociability includes agent interaction. Since a multiagent system is a distributed system where system control is distributed and agents are loosely coupled, information exchange above the physical layer must be incorporated within the individual agents. Modern message exchange software follows a layers pattern as well, referred to as a protocol stack. Fig. 1 includes the protocol stack. While interaction is primarily reflected on layer L5, layer L2 must account for the sequence of message exchanges between the agents, and layer L3 for the common understanding among the agents.

3. A DEPENDABILITY MODEL

3.1 Failures and dependability

We base our a dependability model on (some of) the definitions of Laprie [1]. Computer system **dependability** is the *quality of the delivered service* such that *reliance* can *justifiably* be placed on this service. The **service** delivered by a system is the system behavior *as it is perceived* by another special system(s) interacting with the considered system: its user(s). The service **specification** is an agreed description of the expected service. A system **failure** occurs when the delivered service deviates from the specified service.

The cause—in its phenomenological sense—of any failure is a (unintentional) **fault**. We should make sure that only very few faults occur in an agent. We speak of *fault avoidance* as the col-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

AAMAS'06 May 8–12 2006, Hakodate, Hokkaido, Japan.
Copyright 2006 ACM 1-59593-303-4/06/0005 ...\$5.00.

lection of all measures that reduce the number of faults. Upon occurrence, a fault creates a **latent error**. A latent error becomes an **effective error** when it is activated in the course of providing a service. Effective errors can be subject to treatment. If left untreated they are observed as failures.

3.2 Dependability model for a layer

A *dependability model* describes the service, the effective errors occurring within the system, which ones are dealt with and how (*error processing*), and what the ensuing failures are. Although we only consider faults endogenous to a multiagent system, from the perspective of the individual agent some may be due to exogenous failures, henceforth called external faults. The other faults are internal. An *internal* fault may be due to a programming fault or faulty parameter settings. *External* faults are either *infrastructure failures*, i.e., failures of the hardware and software the agent depends on to function (basically layers L1 and L2), or *peer failures*, i.e., agents on the same level of service that fail to deliver on a request due to connection loss, their own total failure, or inability to reach the desired objective due to unfavorable conditions (*unfavorable outcome*).

Error processing may at best mask the fault, i.e., the state reached after servicing the request is identical to the *regular state*. Next to best, the state is still the *old state*, i.e., the fault simply causes the request to be ignored. A bit worse, servicing the request may reach a state that while no longer meeting the specification may still allow the service requestor a suitable continuation of its work (*sustainable state*). At worst, the outcome may be a state that is plainly *incorrect* if not outright disastrous.

Our goal is to relate the dependability model to the layered reference architecture. Fig. 2 shows two layers where the lower layer provides a service requested by the upper layer. The figure summarizes which faults may occur, what kind of error processing may take care of them, which ones must be passed up to the next higher layer and which of these can receive special treatment there.

4. LAYERED DEPENDABILITY MODEL

4.1 Compressed model

Fig. 2 demonstrates that error processing cannot always be confined to a single layer. Instead it must often be spread to higher layers where a wider context is available, e.g., a sequence of statements rather than a single one, or a larger database. Consequently, a layered architecture should help to organize and localize the various dependability issues. We use the layered reference architecture of Fig. 1 for organizing the dependability of agents.

The result is shown in Fig. 3. It uses an abstraction of the dependability model of Fig. 2. A layer can be in one of two operation modes. During normal activity the component receives service requests from layers higher up, eventually accomplishes them using the services of the next lower layer components and returns normal responses. It may also draw on the services of other components on the same layer or receive service requests from such components.

If the lower layer fails to service the request, this can be interpreted on the current layer as the occurrence of a fault. The mode changes to abnormal activity where the fault is analyzed and processed along the model of Fig. 2. Propagated errors are signaled as a failure to the next higher layer. Otherwise the layer returns to normal mode and tries to resume normal operation.

Abnormal activity mode may also be entered due to abnormal events on the same layer. One such event is a malfunction within the same component—an internal fault. A typical mechanism for signaling such an event is (in Java terminology) by throwing ex-

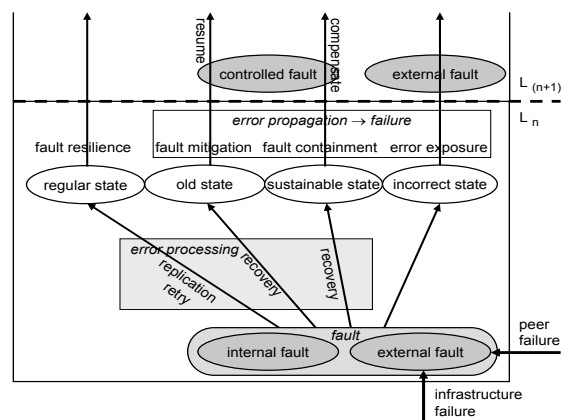


Figure 2: Dependability model: Faults, error processing and responsibilities

ceptions. Exceptions can also be used to model a situation where a fault was not activated further down and thus remains unrecognized while crossing a layer in normal mode and moving upwards as a normal service response. Only on a layer higher up the fault is detected as such. Another kind of abnormal event is peer failures. If they are recognized as such they are being dealt with, otherwise they propagate upwards as part of a normal service response only to induce an exception higher up.

For the rest of this section we give a structured overview of the various faults that may occur on each layer, and short summaries of some of the approaches to cope with them (Fig. 3).

4.2 Layered model

All failures occurring on Layer L1 are infrastructure failures. The individual agent is affected by hardware, operating system, and data transmission failures. Operating systems have a built-in dependability so that only a fraction of faults such as excessive waiting or an unsuccessful disk operation are propagated upwards with a detailed description of the cause. Similarly, many of the faults typical for classical data communication are internally resolved by any of the numerous protocols so that only few failures are propagated upwards, such as network delays, connection loss, non-correctable transmission failures, unreachable destination. Some faults may cause a catastrophic failure, though. It cannot be recognized by the affected agent but will only be detected by collaborating agents ("node is down!").

Layer L2 covers specialized infrastructure services that are usually provided by agent frameworks or middleware. Typical interoperability services concern message handling and exchange, trading and naming, or agent mobility. For example, if an uncorrectable data transmission failure occurs on layer L1 resulting in a message loss, layer L2 may observe a time-out before the expected message has been received, and an exception method may invoke a retry by requesting a resend of the original message. A fault internal to layer L2 may be an invalid agent registration which is best be treated by notifying the upper layers of the failure to execute the needed agent service, where most probably the coordination layer (L5) would have to handle it by finding an alternative peer.

Layers L3 and L4 are tailored towards the domain semantics, with L3 dealing foremost with static aspects and layer L4 with the dynamics. Consequently, layer L3 mostly deals with peer failures in connection with ontologies, e.g., a mismatch in the meaning attributed to a certain term, or different coding into and decoding from the transport syntax.

Layer 4 deals with agent behavior and is in the best position to

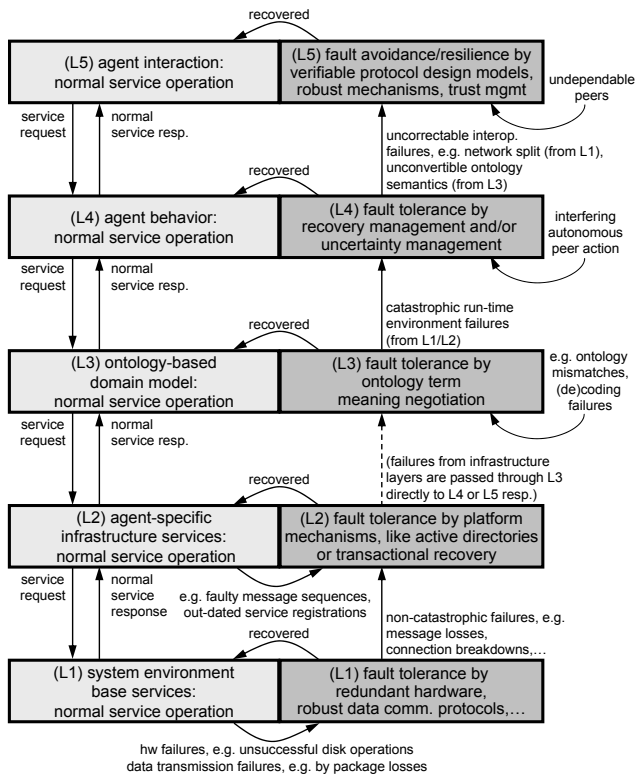


Figure 3: Dependability in the layered agent architecture

process the errors exposed from below. More precisely, L4 should be the layer of choice for all faults whose handling requires knowledge on the domain, the goals and the past interaction history. Consequently, most of the error processing will fall under the heading of fault resilience. This is true even for catastrophic infrastructure failures. Suppose the agent temporarily loses control when its platform goes down, but regains it when the platform is restarted. Recovery from catastrophic failure is usually by backward recovery on the platform layer. The resulting fault containment should now be interpreted in light of the agent's prior activities.

Layer L5 must ensure the dependable performance of the entire multiagent system. We refer to the coordinated interaction of agents as a conversation. Conversations follow a—usually predefined—conversation protocol. At runtime conversations are threatened by failures that are passed upwards from the lower layers. The infrastructure layers may pass up network splits and unreachable peers, the ontology layer semantic disagreements, the agent behavior layer unforeseen behavior of peer agents. Layer L5 may itself discover that peers respond in mutually incompatible ways. Techniques of fault resilience seem essential if the user is not to be burdened. For example, the agents may search for alternative services that are either very similar to the one originally needed, or they may attempt to agree on a new protocol with different peer roles or peers. A careful design should identify which errors should be left to the user and be passed as failures to what one could conceptually see as an additional “user” layer.

5. SPECIAL TECHNIQUES

Our group has extensively experimented with dependability issues in software agents. Below we give a few examples.

An abstraction of behavior that includes recovery is the transaction. The transactional abstraction of a conversation is the dis-

tributed transaction. Fault containment by recovery is then implemented via a 2-Phase-Commit (2PC) protocol, with a single coordinator as the initiator and any number of clients [6]. Hence, conversations spawn local transactions in the participating agents, that are coordinated by the initiating agent. (For technical details see [5]). The drawback is that 2PC robs the agents of some of their autonomy. Since 2PC ignores any transaction semantics, transactional conversations are a layer L2 technique.

Autonomy is preserved if only the individual agents are transactional but not the conversation as a whole. Hence, after recovery the agents are uncertain of the state of the conversation, they just carry beliefs. We modeled individual beliefs as fuzzy propositions over a discrete set of values and introduce a gradual consistency degree over time. Fuzzified versions of belief revision operators allow to modify belief sets while observing a consistency prescribed degree. Fault mitigation by belief revision is a compensation technique for layers L3 and L4.

If one associates compensation with an entire conversation and, hence, an entire group of agents, the conversation must be understood to some detail. Consequently, such an approach is located on layers L4 and L5. Nested transactions for the agents fit the bill. They include diversified error processing such as local or global abort, backward recovery and retry, forward recovery and compensation. The synchronization that comes with a conversation is directly built into the nested transactions for the agents [4]. The drawback of this layer L4/L5 approach is the missing separation between agent behavior and conversation.

6. CONCLUSIONS

The paper demonstrates that a layered reference architecture for agents offers a framework for systematically allocating failure occurrences and the ensuing dependability mechanisms. Thus the architecture provides the means to include in the design of multiagent systems dependability right from the beginning. Our own work (funded by Deutsche Forschungsgemeinschaft) seems to suggest that the approach is indeed viable, but also indicates that dependability techniques for agents still raise intricate issues.

7. REFERENCES

- [1] J. C. Laprie. Dependable computing and fault tolerance: concepts and terminology. In *15th IEEE Symposium on Fault Tolerant Computing Systems (FTCS-15)*, pages 2–11, 1985.
- [2] P. C. Lockemann, J. Nimis, L. Braubach, A. Pokahr, and W. Lamersdorf. Architectural design. In S. Kirm, O. Herzog, P. C. Lockemann, and O. Spaniol, editors, *Multiagent Engineering Theory and Applications in Enterprises*, International Handbooks on Information Systems. Springer Verlag, 2006 (to appear).
- [3] P. C. Lockemann and R. Witte. Agents and databases: Friends or foes? In *9th International Database Applications and Engineering Symposium*, pages 137–147, 2005.
- [4] K. Nagi. *Transactional Agents: Towards a Robust Multi-Agent System*, volume 2249 of *LNCS*. Springer, 2001.
- [5] J. Nimis and P. Lockemann. Robust multi-agent systems: The transactional conversation approach. In *1st International Workshop on Safety and Security in Multiagent Systems*, 2004.
- [6] G. Weikum and G. Vossen. *Transactional information systems: theory, algorithms and the practice of concurrency control and recovery*. Morgan Kaufmann, 2002.
- [7] M. J. Wooldridge. *An Introduction to MultiAgent Systems*. Wiley, 2002.