BAMBERGER BEITRÄGE

ZUR WIRTSCHAFTSINFORMATIK UND ANGEWANDTEN INFORMATIK

ISSN 0937-3349

Nr. 67

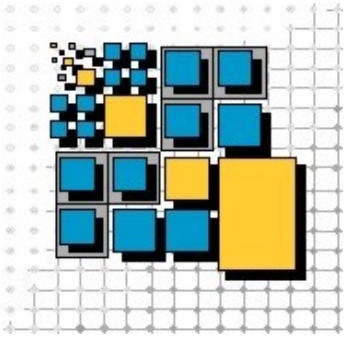
Decentralized Reputation Management for Cooperating Software Agents in Open Multi-Agent Systems

Marco Fischer, Andreas Grünert, Sebastian Hudert, Stefan König, Kira Lenskaya, Gregor Scheithauer, Sven Kaffille, and Guido Wirtz

April 2006

FAKULTÄT WIRTSCHAFTSINFORMATIK UND ANGEWANDTE INFORMATIK

OTTO-FRIEDRICH-UNIVERSITÄT BAMBERG



Distributed and Mobile Systems Group

Otto-Friedrich Universität Bamberg

Feldkirchenstr. 21, 96052 Bamberg, GERMANY

Prof. Dr. rer. nat. Guido Wirtz

http://www.uni-bamberg.de/en/fakultaeten/wiai/faecher/informatik/lspi/

Due to hardware developments, strong application needs and the overwhelming influence of the net in almost all areas, distributed and mobile systems, especially software systems, have become one of the most important topics for nowadays software industry. Unfortunately, distribution adds its share to the problems of developing complex software systems. Heterogeneity in both, hardware and software, concurrency, distribution of components and the need for interoperability between different systems complicate matters. Moreover, new technical aspects like resource management, load balancing and deadlock handling put an additional burden onto the developer. Although subject to permanent changes, distributed systems have high requirements w.r.t. dependability, robustness and performance.

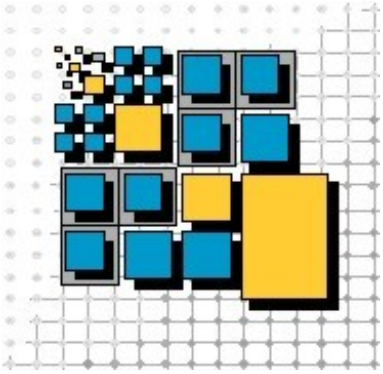
The long-term common goal of our research efforts is the development, implementation and evaluation of methods helpful for the development of robust and easy-to-use software for complex systems in general while putting a focus on the problems and issues regarding the software development for distributed as well as mobile systems on all levels. Our current research activities are focussed on different aspects centered around that theme:

- *Robust and adaptive Service-oriented Architectures*: Development of design methods, languages and middleware to ease the development of SOAs with an emphasis on provable correct systems that allow for early design-evaluation due to rigorous development methods and tools. Additionally, we work on approaches to autonomic components and container-support for such components in order to ensure robustness also at runtime.
- *Agent and Multi-Agent (MAS) Technology:* Development of new approaches to use Multi-Agent-Systems and negotiation techniques, for designing, organizing and optimizing complex distributed systems, esp. service-based architectures.
- *Peer-to-Peer Systems*: Development of algorithms, techniques and middleware suitable for building applications based on unstructured as well as structured P2P systems. A specific focus is put on privacy as well as anonymity issues.
- *Context-Models and Context-Support for small mobile devices*: Investigation of techniques for providing, representing and exchanging context information in networks of small mobile devices like, e.g. PDAs or smart phones. The focus is on the development of a truly distributed context model taking care of information reliability as well as privacy issues.
- *Visual Programming- and Design-Languages*: The goal of this long-term effort is the utilitization of visual metaphores and languages as well as visualization techniques to make design- and programming languages more understandable and, hence, easy-to-use.

More information about our work, i.e., projects, papers and software, is available at our homepage. If you have any questions or suggestions regarding this report or our work in general, don't hesitate to contact me at guido.wirtz@wiai.uni-bamberg.de

Guido Wirtz

Bamberg, April 2006



Decentralized Reputation Management for Cooperating Software Agents in Open Multi-Agent Systems

Marco Fischer, Andreas Grünert, Sebastian Hudert, Stefan König, Kira Lenskaya, Gregor Scheithauer, Sven Kaffille, and Guido Wirtz

Lehrstuhl für Praktische Informatik, Fakultät WIAI

Abstract Multi-Agent Systems (MAS) promise a new advance in distributed computing. In MAS autonomous software agents flexibly cooperate, coordinate and compete to provide the desired function(s) of such a system. If some components of a MAS fail or do not provide the desired functionality, the system is expected to autonomously deal with these situations. It is desirable to reduce occurrences of such situations by selecting trustworthy cooperation partners before cooperating with them. This becomes even more desirable and important in an open MAS where arbitrary heterogeneous software agents, deployed by different parties, participate in the MAS, as it becomes more likely that these agents fail or try to exploit the MAS for their own purposes without reciprocation. In order to monitor agent behavior and enable selection of trustworthy cooperation partners, trust or reputation management services can be applied. As there is no central control in an open MAS and it is completely distributed these services itself have to be distributed. This paper proposes a fully distributed reputation management service for open MAS based on peer-to-peer technology (especially distributed hash tables). A Java-based implementation of that service, which is intended as a plug-in for multi-agent platforms, is also described.

Keywords Reputation management, trust, software agents, cooperative problem solving, multi-agent systems

Contents

1 Necessity for trust in open Multi-Agent Systems **1**

1.1 Multi-Agent Systems 1

1.2 Cooperative Problem Solving in MAS 2

1.3 Multi-Agent platforms 3

1.4 Trust and Reputation 4

2 Requirements on Reputation Management **6**

2.1 Basic requirements 6

2.2 Derived requirements 7

2.3 Attacks on Reputation Management 8

3 Architecture **9**

4 Protocols and counter measures against attacks **11**

4.1 Chord overlay protocols 11

4.2 Identification of agents and agent platforms 13

4.3 A Platform joins 14

4.4 A Platform leaves 15

4.5 A Platform is excluded 15

4.6 An agent registers 16

4.7 An agent de-registers 18

4.8 An agent rates other agents 18

4.9 Reputation of an agent is retrieved 21

4.10 Checking consistency of data 21

5 Implementation in Java **22**

5.1 Interface of the plug-in 23

5.2 Implementing the architecture with Java 24

5.2.1 Layer 3 24
5.2.2 Layer 2 24
5.2.3 Layer 1 27
5.2.4 Layer 0 29

6 Conclusion **33**

Bibliography **35**

A List of previous University of Bamberg reports **37**

List of Figures

1 The three 'dimensions' of trust. 4
2 Architecture of reputation management service. 10
3 Chord routing with help of finger tables. 12
4 The metric strategy pattern 25
5 Architectural overview of Open Chord. 31

List of Tables

1 Attacks: A Platform joins 15
2 Attacks: a Platform is excluded 16
3 Attacks: an Agent registers 17
4 Attacks: an Agent rates other agents 20
5 Attacks: Reputation of an agent is retrieved 21
6 Attacks making Consistency Check necessary 22

List of Abbreviations

CPS	Cooperative Problem Solving
DHT	Distributed Hash Table
JVM	Java Virtual Machine
MAS	Multi-Agent System
P2P	Peer-to-Peer
PGP	Pretty Good Privacy
URL	Uniform Resource Locator

1 Necessity for trust in open Multi-Agent Systems

This paper describes a fully distributed reputation management scheme to establish trust in open Multi-Agent Systems (MAS). Reputation management can be used as a foundation for agents to select trustworthy cooperation partners. For this purpose the paper is organized as follows. This section motivates why distributed reputation management is necessary in open MAS and describes the characteristics of open MAS. The second section exemplifies, which requirements reputation management for open MAS must meet, while the third section deals with the architecture of the proposed reputation management. Section four explains the protocols employed to implement distributed reputation management. The succeeding section describes how the proposed reputation management has been implemented in Java. The last section summarizes the results of this paper and addresses open issues and future extensions of our approach.

1.1 Multi-Agent Systems

Agents seem to be the next promising paradigm in Software Engineering for complex distributed systems. Currently there is no consensus on a definition what an agent is, but most definitions agree that an agent has to be autonomous, situated in an environment and must be able to percept its environment and react to and change the environment [JSW98]. Furthermore a single agent can be reactive or a complex deliberative entity. A reactive agent autonomously takes actions based on perceptions just made from its environment, whereas a deliberative agent can be characterized as autonomously planning its (current and future) actions based on an internal environment model.

A MAS is according to [JSW98] „a loosely coupled network of problem solvers that work together to solve problems that are beyond the individual capabilities or knowledge of each problem solver". These MASs can be constructed (i) with help of top-down/divide-and-conquer approach or by (ii) a bottom-up approach by composing individual agents or groups of agents with different capabilities and knowledge. For (i) all problem solving and communication strategies become a hard-wired integral part of a MAS. Whereas (ii) requires coordination and negotiation between agents to achieve a solution for their problem(s). It must be possible for agents themselves and users of the MAS to define new goals as there is no hard-wired overall system goal provided at design time of the MAS. A MAS can be described by the following characteristics:

- Each agent has limited capabilities and knowledge to solve problem(s) for itself
- There exists *no global control*.
- Data storage is *decentralized*.
- Computation is *asynchronous*.
- The configuration of a MAS is *heterogeneous* regarding the agents.

- A MAS can be *closed or open*. In closed MAS only agents specified at design time can take actions as described by the design. Closed MAS are most likely the result of a top-down/divide-and-conquer design appraoch. In a open MAS agents can leave and enter at runtime as they desire.
- A MAS is expected to operate in a highly *dynamic environment*, which on the one hand may result from its openness and on the other hand the nature of its environment.

In open MAS, that are for example deployed on the internet, agents of different designers may enter the system and have their own (possibly selfish) goals. Because of these agents and their autonomy new patterns of cooperation and behaviors may emerge. It is important to ensure that autonomy of agents and emergence of new features of a MAS do not destroy or disturb the functions and goals, which the MAS was designed to achieve. In order to ensure the proper function and achievement of goals of agents, that are already part of the MAS, mechanisms to prevent disadvantages for these agents have to be implemented.

1.2 Cooperative Problem Solving in MAS

In MAS agents have to cooperate to achieve their goals, as single agents often have limited resources, capabilities, or knowledge.

Cooperation in MAS can be structured with help of the Cooperative Problem Solving (CPS) process ([WJ99], [DKV03]). CPS is divided into four phases: recognition, team formation, plan formation, and team action.

1. *Recognition*: At least one agent must recognize that there is the need or potential for cooperation. Furthermore a group of agents, that can cooperate must be identified.
2. *Team Formation*: In this phase the agents try to let all other agents know that there is a cooperation possible. If this phase is successful a potential team has been found.
3. *Plan Formation*: Now the agents try to find a common plan to achieve their goal(s). If they find and agree upon a common plan they engage into a joint commitment. They promise each other to carry out their plan. If there is no agreement on a common plan the cooperation fails.
4. *Team Action*: In this last phase the agents carry out the plan agreed upon before.

This phases have not to be processed sequentially. CPS is rather an iterative process and some of the phases may be left out. If one phase fails, agents can go back to an earlier phase.

For CPS to be successfully carried out, agents must reveal their true goal(s). They must fulfill the tasks, that have been assigned to them as desired by the whole team. It is assumed that all agents are sincere and reveal their true goal(s). This cannot be assumed in an open MAS, as the developers of agents may have bad intentions. There may for example exist some agents, that just want to disturb the correct function of a MAS (denial of service), or agents that just

pretend to cooperate to achieve their goals with less costs, but do not fulfill their share (free riding).

In order for CPS to be successful in open MAS, agents must have a basis for the decision with which agents to cooperate. So that they in phase two (team formation) can decide which agents should be in the team as they will most likely fulfill their assigned tasks. In phase three it is helpful to have knowledge about the reliability of agents regarding the tasks they can carry out in order to assign them to the most reliable agents. After the fourth phase of CPS has been carried out in an open MAS a fifth phase, which records the experiences agents make with each other should follow. One possibility explored in MAS research to identify and exclude agents that behave badly is Trust Management ([CF98], [WS00], [GLd02], [SS02a]). Before section 1.4 describes the properties of trust and reputation, the following section describes some services that are required for open MAS.

1.3 Multi-Agent platforms

For execution of software agents runtime environments and services are required. These services comprise - among other things - life-cycle-management, directory services, and communication services [FIP02], to find and communicate with cooperation partners for CPS. In this paper a runtime environment providing such services is called an agent platform (or just platform).

To provide an open MAS it must be possible that agent platforms on different machines can be connected to facilitate cooperation between the agents residing to these platforms. It must also be possible that new platforms and new agents can enter the MAS at runtime.

Because of this dynamic nature of open MAS it is reasonable to base a MAS on Peer-to-Peer (P2P) technologies, as these kind of systems provide mechanisms to dynamically handle arrival and departure of new machines in a network. The services (e.g. discovery services) required by agents are then provided by all the platforms that are connected within the P2P network to equally distribute load and make the whole network scalable as with the arrival of new platforms, these platforms will also provide new resources.

One further service that should be provided by agent platforms to better support CPS is a service to keep track of agent behavior. This service should also be distributed equally among platforms participating within the P2P network, but should be seen as a logical central unit. On the one hand distribution of such a service among platforms ensures that there exists no single point of failure or a single entity that can be attacked. On the other hand distribution facilitates that platforms deny to fulfill their duty in provision of the service. All the same in an open MAS where agents and platforms of different parties can leave and enter as they desire a fully distributed service is more desirable so that every participant has to provide the same resources to the open MAS. As mentioned in the previous section one possibility explored in MAS research to identify and exclude agents that behave badly is trust management, which can be based on so-called reputation management. As current platforms have no built-in service for trust or reputation management, developing such a service is the goal of our work. Before defining the requirements on such a system, some required terms regarding trust and reputation are defined in the next subsection.

1.4 Trust and Reputation

Trust can be used as a foundation for reasoning about with whom to interact in situations where only partial information about the partner is available and there is a risk that the interaction may be harmful. Trust depends on situation and the context, in which it is validated [CSG+03]. Trust between two individuals is not necessarily symmetric. In MAS the context, in which an agent can be trusted, can be defined by using the role an agent plays [CBG02].

The primary source used for building trust to someone is a subjective image [SS02b] one can have of the other party. If such an image is not available, one has to resort to other information sources as e.g. a reputation management system which provides reputation values. Reputation management [ZM00] facilitates estimation of trust between agents based on a history of interactions.

Reputation management is mostly realized with help of a central trusted entity, which stores reputation values for pairs of agents that have interacted before. If such a central entity is present, an agent can ask it (instead of asking all agents known by it) for all interactions, in which the agent under consideration has been involved, and how other agents rated these interactions.

In order to make reputation and trust interpretable and computable by software it is often represented by numerical values (e.g. real numbers between 0 and 1). The reputation value for a single agent is then computed with help of a so-called reputation or trust metric [GH04].

Reputation information about an agent can be seen by all other agents including the affected agent itself. Therefore the agent can estimate if other agents have trust in it and how it is rated compared to other agents. This will give the agent an incentive to keep its reputation as high as possible by behaving well in any interaction. Ratings of agents that rated others, but have already left the system are still maintained by the reputation management. According to

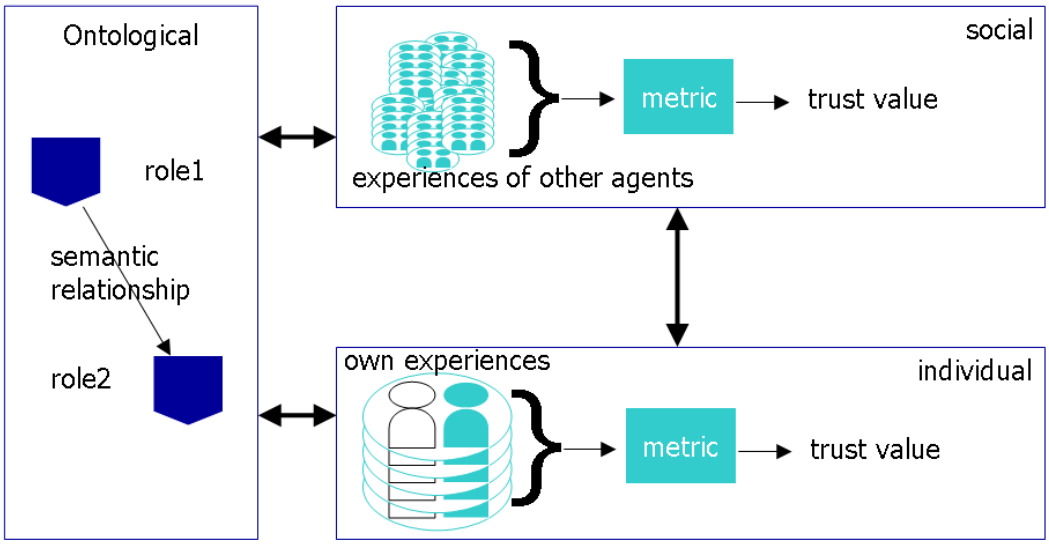


Figure 1: The three 'dimensions' of trust.

[SS02b], reputation can be structured in three dimensions [1]:

- Individual dimension: This dimension deals with the direct interaction between two agents. Through the direct interactions agents get the most reliable (although subjective) information about others.
- Social dimension: If an agent had no direct contact with a potential interaction partner, it has no information about it and therefore it may try to get information about other agents by asking other agents. While doing that it must be considered that the information gathered from other agents is also subjective.
- Ontological dimension: it can be used in these other dimensions to calculate reputation based on ontological relationships between behaviors of agents. Trust is related to a single behavioral aspect (context). The ontological dimension makes it possible to describe relations between different aspects to allow computation of more complex trust values.

Figure 1 shows possible relationships between the three 'dimensions' of trust. To compute a trust value for an agent different sources of information may be combined. E.g. an agent can retrieve reputation values from other agents (social) regarding one cooperation partner in a certain role A and combine these with the its experiences (individual) it made with that agent playing role A and playing ohter roles that are related to A (ontological).

In an open MAS the individual dimension should be supported by each single agent itself. The social dimension can be realized with help of a concept called web of trust or with help of a central entity that keeps a history of agent behavior. In order to determine if an agent can trust a new cooperation partner, in a web of trust it uses its relationships to other agents it knows to query for experiences the other agents eventually made with the potential partner. For this purpose it must also trust in the recommendations of its acquaintances and keep track of these.

In order to support the social dimension of trust and make it easier to compute reputation for an unknown cooperation partner a central entity seems to be more promising. It saves reputation values for pairs of agents that have interacted before. If such a central entity were present, an agent could ask a single entity (instead of asking all agents known by it) for all interactions, in which the affected agent has been involved, and how the other agents rated these interactions. Such a foundation for creation of trust between agents based on historical interactions is called reputation management ([ARH00], [ZM00]). Reputation information about an agent can be seen by all other agents including the affected agent itself. Therefore the agent can estimate if other agents have trust in it and how it is rated compared to other agents. This will give the agent an incentive to keep its reputation as high as possible by behaving well in any interaction. Ratings of agents that rated others, but have already left the system are still maintained by the reputation management, which amplifies the availability of information about agents.

The ontological dimension of trust can be addressed if agents reputations refer to the role(s) an agent played in CPS. A role according to [PV02] 'cluster types of behavior into a meaningful unit that contributes to an overall goal'. These roles can provide the context for the trust that

[1] In opinion of the authors the term 'dimension' does not fit very well, as the individual, social, and ontological aspects of trust are not always orthogonal to each other. How they are combined may depend on the application domain.

an agent can have in another agent. There may exist relations between roles that allow to infer trust from the behavior of one agent within one role from the behavior in another role. For example we may trust a car mechanic more to fix a bicycle than a clerk.

Our approach assists the social dimension by providing a reputation management service which itself is completely distributed. The ontological dimension can be addressed with help of the roles an agent plays, but is not covered in this paper. In our approach each single agent is responsible for the individual dimension on its own.

2 Requirements on Reputation Management

This section deals with the requirements on reputation management in open MAS based on the characterizations of MAS and reputation management from the previous section. For this purpose the next subsection describes requirements that are directly imposed by the remarks in the previous section. The second subsection explains the requirements additionally derived from these basic requirements. The last subsection deals with attacks that can be performed by agents or agent platforms to disrupt or abuse reputation management. Some of these attacks may result from the requirements described in this section.

2.1 Basic requirements

A service for an agent platform that provides reputation management must be developed. The reputation management service must enable agents to rate and request the reputation of other agents. Agents must be free to leave and enter the MAS at any time. If an agent leaves and it has gained a certain reputation it must be possible for the agent to maintain this reputation and regain it, when it re-enters the system. This must be possible from any agent platform.

As the MAS is distributed itself and there exists no central global control, the platforms should be organized into a P2P overlay network to facilitate an open network, where platforms (and the agents executed by them) can join and leave at runtime. Within these network the load and resource consumption of the reputation management service should be equally distributed among participating platforms, so that every participant of a MAS has to provide resources for reputation management. The load and resource consumption must be scalable with regard to the number of participating agent platforms. By distributing reputation management in a P2P-fashion it is more likely that is scales with the number of participating platforms and agents.

In order to facilitate reputation management agents must be uniquely identifiable, so that they can be rated by other agents. Therefore an identification and authentication mechanism must be available.

As reputation is context-dependent it must be possible to save reputation information for certain contexts. These contexts are provided by CPS processes that are conducted by the agents. It must be possible to rate the agents with help of the role(s) they play while cooperating with

others.

As there are already some implementations of multi-agent platforms available, these platforms must be enabled to use the proposed reputation management. Therefore it must be developed as kind of a plug-in for existing platforms. If in the following sections the responsibilities of a platform are mentioned, these responsibilities are taken over by the plug-in if not stated otherwise.

2.2 Derived requirements

To implement the requirements stated in the previous section some further requirements have to be imposed. A P2P scalable and load balanced overlay network that provides the possibility to store reputation data and information about agent identities must be available. For this purpose a distributed hash table (DHT) can be used.

In a DHT data is stored on nodes (here these nodes are the agent platforms) that participate in the P2P overlay network. Any node can manipulate the data which is stored on it. Therefore mechanisms to ensure data integrity and consistency must be provided.

As agents can rate other agents, they influence how other agents judge about trustworthiness of these agents. Therefore it must not be possible to rate others when there has been no interaction between the agent providing the rating and the agent being rated.

Mechanisms must be developed that provide identification and authentication based on well-understood cryptographic concepts. Furthermore mechanisms to handle agent arrival and departure, where agents provide existing reputation data concerning them must be provided. These mechanisms must make sure that the provided data is valid reputation data.

As agents could cooperate with the platform that is hosting them, this platform must not alone be responsible for storage of data about these agents, because it is very likely that it removes data that is bad for the reputation of its agents. If other platforms store data of agents of one platform this platform must not be able to manipulate the data. Therefore it must be possible to check the consistency of these data by the platform that hosts an agent or by the agent itself. A platform storing data also should not be responsible for data alone, as it cannot only manipulate data, but also remove it. Therefore reputation data should be distributed among more than one platform.

As can be seen from this facts, trustworthiness of platforms themselves has to be tracked with help of a platform reputation. Therefore reputation management must also manage the reputation of agent platforms and make sure that only trustworthy platforms participate in the P2P network.

In order to assure that agents can only rate others if there has been a transaction, transactions must be announced to the reputation management service. The reputation management service of an agent platform must create a unique identifier for this transaction that can only be used once to rate other agents. As the platform can be cooperating with the agents it is hosting it cannot alone be responsible for creating such an identifier. A mechanism that makes it possible

to create such a transaction identifier by more than one platform must be developed.

To ensure that agents do not request a transaction identifier and they do not give a rating for the requested transaction, a protocol must be developed, which ensures that ratings after a transaction are delivered. Also a mechanism that motivates agents to leave the MAS in an orderly fashion, so that their reputation data can be removed to save storage capacity, must be developed. These two mechanisms can be realized with help of a leasing concept, where agents must lease transactions and the storage of reputation data. This leasing mechanism must be distributed among several platforms. It cannot be realized by the platform alone, that hosts the agent for which it manages a lease, as the platform may cooperate with the agent.

All these mechanisms require that platforms can communicate with each other in order to coordinate their activities. The integrity and reliability of this communication must be secured.

Different domains may require different representations for reputation values and different reputation metrics. Therefore it should be possible to easily exchange the reputation value representation and reputation metric.

2.3 Attacks on Reputation Management

Beyond these requirements the distributed reputation management service must prohibit attacks that are possible on every reputation management. Furthermore attacks that can be conducted because of the distributed nature (derived from the requirements above) of the reputation management service have to be prevented. These attacks can be divided in two categories. On the one hand attacks can be executed by agents, on the other hand they can be executed by agent platforms. These attacks are listed below[2].

Attacks by agents:

AA1 Pretending to be trustworthy: An agent pretends to be trustworthy for a couple of transactions to create a good reputation value and then changes its behavior to exploit others.

AA2 Faked transactions: Agents rate each other without a transaction.

AA3 Faked ratings: An agent cooperates with other agents to get a high reputation value.

AA4 Harmful rating: An agent rates another agent with a bad reputation value without reason.

AA5 Change of identity: An agent leaves because of a bad reputation and creates a new identity to get rid of the bad reputation.

AA6 Avoid rating by other agents: An agent may try to cooperate with other agents without being rated afterwards.

Attacks by agent platforms:

[2]Please note, that some of these attacks cannot be handled completely by a reputation management service alone

PA1 Refusal of service: A platform does not save or it deletes reputation data of an agent.

PA2 Reputation data manipulation: A platform manipulates existing reputation data of an agent.

PA3 Fake of reputation data: A platform creates fake reputation data for an agent.

PA4 Routing attack: A platform tries to manipulate the underlying peer-to-peer routing mechanism to make reputation data unavailable.

PA5 Denial of Service: A platform tries to make reputation data unavailable by flooding other platforms with unnecessary requests.

PA6 Non-trustworthy platform participates: A platform that has been identified as non-trustworthy, as it tried to perform some or one of the attacks described above, is still or again trying to participate in reputation management.

3 Architecture

This section provides a short overview of the proposed reputation management scheme for agent platforms. It explains the architecture of our reputation management, its single components and their responsibilities.

The reputation management is built as a plug-in and is to be used as a basic service of agent platforms. Its features are invoked by agent platforms using the interface of the so-called Trust Service, either direct or through an agent running on the agent platform. The logic implemented by the plug-in is encapsulated and only accessible through an interface. Thus, the complexity of the application is not visible to the agent platform, the agents and their programmers.

The plug-in consists of four different layers. Each layer performs different kinds of tasks and provides an interface to the layer above. On layer zero, sockets allow communication between different agent platforms. In addition, a distributed hash table (DHT) stores data, such as information about agent platforms, agents and their reputation. This information is accessible from any agent platform inside the network, even though the requested information is not stored on the agent platform itself.

On layer one, the Communication Service is an abstraction for sockets. This allows services on layer two to communicate transparently with plug-ins on other agent platforms. The knowledge which technique is used to perform this communication is not necessary. To prevent unintentional communication from agent platform outside the network, the Communication Service is also responsible for security regarding communication. In consequence, higher-level layers do not face false requests.

Also on layer one, the Data Service provides an interface to perform requests concerning information stored in DHT. Data is stored redundantly on multiple nodes, to address security aspects concerning data storage in an open network. Not a single agent platform is responsible for a date, but multiple agent platforms are responsible to store it. Again, higher-level layers

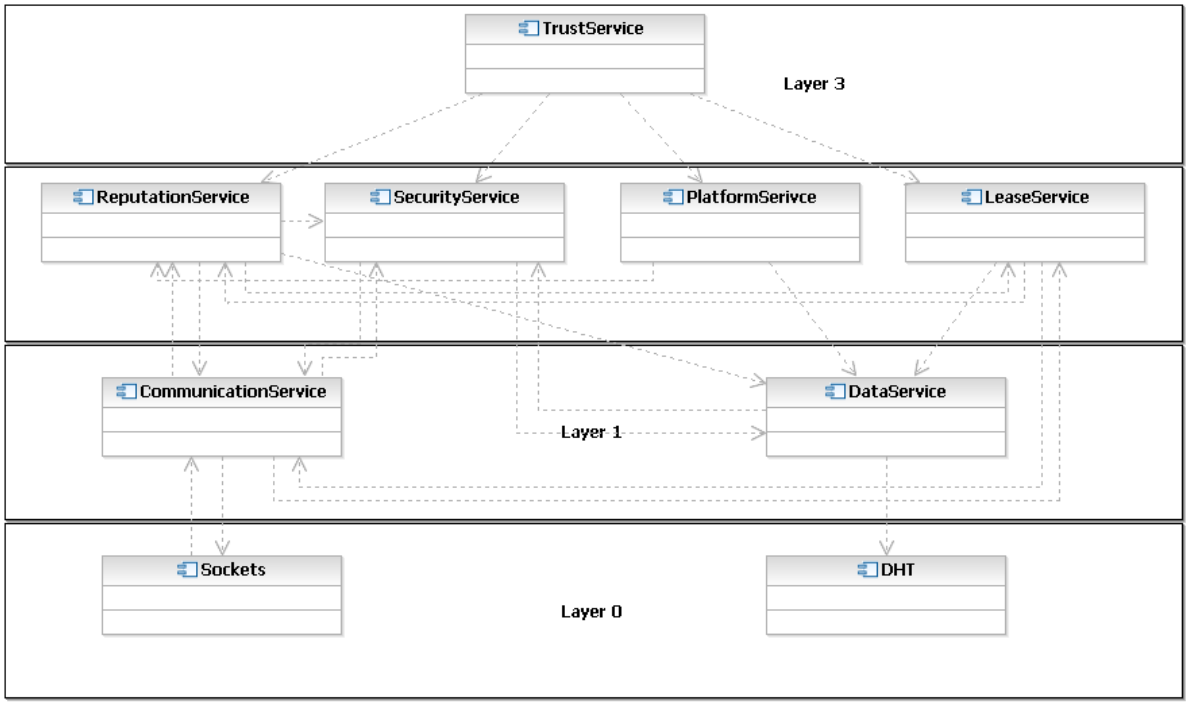


Figure 2: Architecture of reputation management service.

do not need to know which way of persistence technique is used. Thus, the data abstraction grants the possibility to replace the employed implementation of DHT with any other persistence technique. This may be of importance for integrating the plug-in into different agent platforms.

The following services, arranged on layer two, implement the core logic for our reputation management. The Reputation Service conducts the process of rating agents and agent platforms. This means the service calculates ratings for agents and agent platforms using a given rating metric and a reputation-value representation. They are based on data retrieved from by Data Service. Neither the rating metric nor the rating representation is closely attached to the Reputation Service. Due to the abstraction from concrete rating metrics and reputation-value representation, it is possible to use different kinds of rating metrics respectively rating representations which implements a given interface. Several techniques are implemented to avoid false ratings from cooperating agents and wrong ratings from bad agents.

The Security Service on layer two is responsible to provide means for secure communication between agent platforms and secure data storage in the DHT in a way that ensures data cannot be counterfeited. This is done utilizing the techniques of signing and ciphering data using symmetric and asymmetric algorithms. The service provides to the agent platform (and each agent running on it) a private[3] and a public key. Data stored by an agent platform has to be signed by the agent platform itself and to prevent manipulation of this data by that platform by another agent platform. This is done by the private key of the agent platform or respectively of an agent. Communication is secured by symmetrical ciphering of the data stream. In

[3]How private data of agents and an agent platform is being secured is the responsibility of the agent platform itself.

order to do so, a means to create temporary session keys for ciphering the communication between different agent platforms are provided by the Security Service. It is also responsible for validating signatures and deciphering data. Its services are only invoked by others and it performs no action itself.

In order to enter a network, an agent platform uses the Platform Service, which is resident on layer two. The service is used to enter an existing network, create a new network and to leave a network. Moreover, this service addresses data consistency in the DHT. Though agent platforms are not allowed to store data in the DHT without an agreement of another agent platform, data may be altered by cooperating agent platforms. In consequence, an agent platform holding false data needs to be identified and will receive an appropriate rating. Agent platforms with very low reputations are barred from the network.

It is not possible for the plug-in to force agents or agent platforms to use the logout function to leave the network. Thus, it is not always possible to know if an agent is still running/working. This means the plug-in serves reputation data about agents which are possibly not active anymore. Also, agents requesting a ticket in order to rate each other after performing a task, should be motivated to return it to the plug-in, in order to end the process of rating. In order to do so a Lease Service is introduced. A corresponding lease exists for each agent logging into the plug-in. The agent has to inform the plug-in within a certain time span to avoid an expiration of the lease. In case the lease expires, the agent receives a bad rate. If the agent left without notice, its reputation is downgraded each time the lease expires.

4 Protocols and counter measures against attacks

This section describes how the architecture described in section 3 is exploited to implement a distributed reputation management. For this purpose the protocols are presented, which allow to build a network of agent platforms. The next subsection describes the protocols of the underlying DHT, in which most of the employed data structures are stored. Then - before describing the protocols- some data structures and concepts used to identify agents and platforms are explained. The succeeding sections describe how an agent platform can join and leave the network. These are followed by protocols that describe how agents can use the reputation management service. For each protocol -if applicable- the attacks (described in section 2.3) are addressed, which are avoided or for which negative consequences can be alleviated by this protocol.

4.1 Chord overlay protocols

As DHT and P2P foundation for data management in our reputation management system the chord overlay network [SMK+01] is adopted. Chord is a structured P2P infrastructure and in its overlay network the peers are organized into a ring. For this purpose each peer has a unique identifier. This unique identifier is created with help of a hash function from data that uniquely identifies the peer. The unique identifier determines the position of a peer in the ring. Because

of the ring structure of the overlay network every node has a single predecessor and successor.

Data, that is supposed to be stored with help of chord, is also associated with a unique identifier. With help of this unique identifier of data the peer that is responsible to store the data is determined. A peer is responsible to store data with identifiers that are less than or equal to its own identifier and greater than the unique identifier of its predecessor.

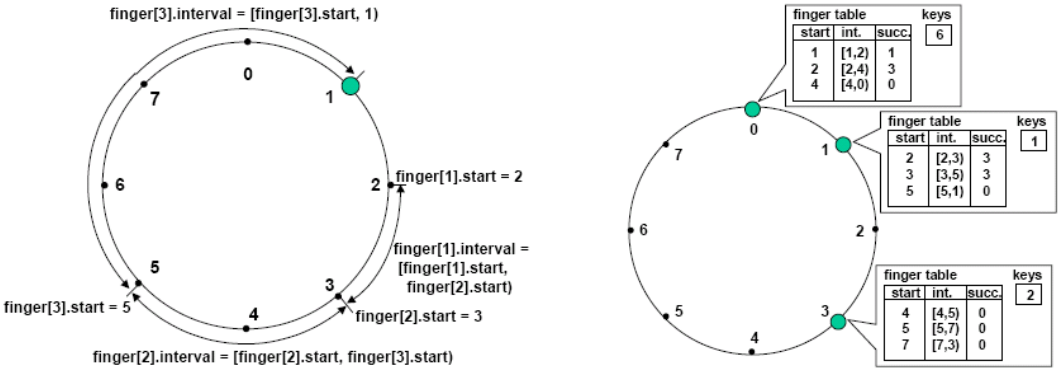


Figure 3: Chord routing with help of finger tables.

In order to facilitate lookup of stored data each peer maintains a routing table, the so-called finger table. The finger table only contains references (addresses of other peers) to peers that are responsible for data with certain identifiers. These identifiers are calculated with help of the unique identifier of a peer (n) and the number of bits of an identifier (m) as follows: $(n + 2^{k-1}) \bmod 2, \quad 1 \leq k \leq m$. Figure 3 shows an example in a 3 bit identifier space. On the left hand side the intervals are depicted, for which node 1 must know other peers, that are responsible for these intervals. The right hand side presents the finger tables of the peers with identifiers `0`, `1`, and `3`.

When a peer wants to store or lookup data with a certain identifier, it has to find the peer which is responsible for data with that identifier. That means the peer that has the smallest greater identifier of all peers compared to the identifier of the data. For this purpose the peer asks the peer from its finger table, whose identifier is the closest preceding identifier of the data identifier, if it knows a peer that is closer to the data identifier. If such a peer exists, this again is asked for a closer predecessor of the data identifier. When a peer is found whose identifier is less than the data identifier and for that the identifier of its successor is greater than the data identifier, the successor of this peer is responsible for the data. This peer can than be requested to store or deliver data which is associated with the data identifier. This procedure ensures that it takes approximately $\log n$ (with n of peers in the network) steps to find a peer that is responsible for an identifier [SMK+01]. It also guarantees to find data that is stored within the DHT given the structure of the chord overlay network can be maintained.

To maintain the structure of chord three tasks have to be periodically performed. The first one checks if the predecessor of a node is still alive. If the predecessor is not alive, the peer temporarily has no predecessor. This is where the second task comes in which checks if the successor of a peer is still available. When the successor of a peer fails, it tries to lookup a new one again. This new successor is then informed that it has the peer, whose successor failed, as new predecessor. With help of this two tasks the predecessor and successor relationships within a chord overlay network are maintained. The third tasks periodically tries to find better

entries for the finger table of a peer by using the lookup functionality of chord. A better entry means the address of a peer whose identifier is closer (but still greater or equal) to the identifier required for a finger-table entry.

When a peer wants to join the chord overlay network a unique identifier is assigned to it and it searches -with help of the procedure described in the previous paragraph- for its successor. When it has found its successor, the predecessor of this peer becomes the predecessor of the joining peer. So it is inserted into the chord ring structure between its successor and the former predecessor of its successor. It therefore takes over the responsibility for all identifiers as described above and copies all data that is stored on its successor for this interval to its local data store.

In order to avoid loss of data a replication scheme to store data on the responsible peer and its successor can be used. To increase data reliability this can be extended by each peer having a list of successors. The peer responsible for a data is then also responsible for replication of this data to its successors.

If a chord overlay network is deployed in an uncertain environment where malicious peers may participate, it cannot be assumed that all peers fulfill their duties. A peer may remove or alter data it is responsible for or it may not replicate it. Therefore it must be ensured that a peer cannot choose an arbitrary identifier to take possession of data with certain identifiers. And if a peer is responsible for certain data it must be possible for it to prove that fact. To avoid that a single peer is responsible for certain data and its replication responsibility for data must be assigned to more than one peer, so that (assumed not all peers are malicious) data cannot be compromised. In our reputation management system responsibility for data is assigned to a number of peers and to ensure consistency of this data periodical checks are conducted. If these checks reveal that data has been altered or removed this is recorded with help of a peers reputation. When the reputation of a peer has reached a low threshold it is excluded from the chord overlay network.

4.2 Identification of agents and agent platforms

In order to facilitate reputation management it must be possible to identify agents and agent platforms. Agents have an identifier, which is unique within the agent platform, that hosts the agent. Therefore the agent can be uniquely identified within a network of agent platforms with help of its unique identifier and the unique identifier of its platform. For identification of agents a data structure containing the identifier of the agent, the identifier of its platform, and the public key of the agent, is provided. This data structure is signed by (at least two) agent platforms that confirm the agents identity and is stored within the DHT.

In a similar manner an identification data structure for an agent platform is provided. It contains the unique identifier, the address that can be used to communicate with the platform, its identifier within the DHT, and the public key of the platform. This data structure is signed by (at least two) other agent platforms, which by this confirm the identity of the platform.

Agents and agent platforms are, as far as the plug-in is concerned, identified by the corre-

sponding data structures in the DHT. To better countermeasure attacks like AA5 and PA6 (refer to section 2.3) not only the technical identifiers of agents and agent platforms should be incorporated into these data structures. They should also incorporate the identity of the owner of the agent or platform to be able to conclude the owner's reputation by analyzing his agents. Thus general policies concerning agents of a particular owner can be specified. The owner's identity should then be certificated by a trusted third party. Then it would be possible to exclude agents or platforms of certain owners. For simplicity in our prototype these considerations are (initially) left out. But they can be easily incorporated by treating owners of agents as special agents with the special role `owner`. Then it would be possible to prevent owners, whose agents or platforms demonstrate untrustworthy behavior, from incorporating more agents and platforms into the distributed reputation management service. For further discussion on identification in reputation management systems see e.g. [CP02].

4.3 A Platform joins

The join of platform to the P2P network is initialized by an agent platform administrator. Therefore it has to be defined whether the new platform should create a new network or attend to an existing network. For a new network no login activity is conducted.

Protocol flow
In order to join the network the plug-in contacts a trusted agent platform (the contact data is commissioned by the administrator) with help of the Communication Service, sends its identification data, and waits for a reply. This platform decides based on the identification data of the new platform if it may join the network. It may neglect joining of the new platform. New generated public and private keys, signed from two trusted platforms, are handed over to the joining platform. The unique identifier of the new platform for its position in the chord DHT is generated from its public key. Therefore all other platforms can verify that the new platform takes over responsibility for the data it is supposed to be responsible for. This data is also stored in the underlying chord DHT to facilitate verification of data from the new platform and authentication of the new platform by already participating platforms.

Afterwards the new platform joins the DHT network by searching for its successor, which ensures (with help of the identification data of the new platform) that the new platform is integrated in the chord DHT structure. The successor verifies with help of the identification data of the new platform (that is available from the DHT, see above) that it really is its predecessor and it is allowed to take over responsibility for data that is smaller than its identifier.

Counter measures against attacks
The first request to a running platform arrives at a Communication Service. In general the request will always be discarded, if the Communication Service identifies the request as a Denial of Service attack. This service also makes sure by involving the Reputation Service, that the platform is trustworthy or completely unknown. If the platform has been excluded from the system, the communication is aborted. Otherwise the request is forwarded to the Platform

PA5	DoS attack
PA6	Non-trustworthy platform participates

Table 1: Attacks: A Platform joins

Service, applying the regular protocol flow shown above.

4.4 A Platform leaves

Departure of a platform is also initialized by the corresponding platform administrator. The platform leaves the network, but all its data (reputation values and identification data) is saved in order to enable the recurrence of the platform.

Protocol flow
The platform informs the network affiliates about the platforms aim. The chord DHT rearranges the responsibility for data which was placed at the leaving node. If the platform leaves the network without notifying its affiliates, the DHT automatically rearranges data and responsibilities analogously after a certain time, because of the maintenance tasks described in section 4.1.

4.5 A Platform is excluded

If a platform has a reputation value below a certain threshold it is not considered as being trustworthy any more. This platform is then excluded from system activities.

Protocol flow
After any platform has noticed, that a given platform is not trustworthy any more, it removes all references to it from its chord finger table and denies possibly incoming requests from that platform. All trustworthy platforms act in this way, so that not trustworthy platforms are excluded from the overlay network. Data and responsibilities of a not trustworthy platform are rearranged as if the platform had left the system. In order to prevent this platform from joining the network again, the platform identification data is kept in a list of (formerly) known platforms. This is also necessary to be able to verify data that has been created by the excluded platform, while it has been trustworthy in the past. The Communication Service of a platform also ceases communication with not trustworthy platforms.

Counter measures against attacks
Participating in the network can occur in two ways: Request-response communication via the Communication Service and saving data in the chord DHT via the Data Service. The former is handled by the Communication Service of the receiving platform, which will block all requests

PA6	Non-trustworthy platform participates

Table 2: Attacks: a Platform is excluded

from no more trustworthy platforms. The latter is prohibited by the mechanisms of the chord DHT described above, which only provides.

4.6 An agent registers

This section describes the use case of registering a new agent. There are two cases of agent registration: in the first a completely new agent registers, in the second case an agent that has participated in reputation management before registers again. Every agent that wants to take part in the reputation management system has to register properly and agents should not cooperate with agents, that are not registered with the reputation management system.

Protocol flow
For each agent a unique identifier for each participating agent is assumed, which is managed by the agent platform that hosts the agent. As each platform also has a unique identifier each agent can be identified unambiguously by a combined identifier consisting of its the platform identifier and its own identifier.

A completely new agent registers with this identifier. The Security Service creates a new key pair for the agent, used signatures during its participation in the reputation management network. To create this key pair an asynchronous ciphering algorithm is used (for the prototype: RSA). After that a new agent-identification data structure (see 4.2) containing the agents identifier and public key, is created and saved in the DHT.

The Lease Service of all nodes responsible to save the identification data structure start a lease for the joining agent. This is considered as successful if at least one node has started the lease correctly, otherwise an exception occurs and the agent cannot register. The timespan used for agent leases is a global constant.

Finally the private and public key, and the timespan for the initiated lease are returned to the agent. Thus the agent has all information it needs for participating, like its key pair and the lease time, and is therefore correctly affiliated with the network.

In case the agent has formerly participated in the network it can return and register providing not only its identifier but also its reputation data, which were provided to it when it left the system.

To register again the Security Service will check if the provided identifier of the agent matches the one for which the reputation data is valid. If the identifier is correct it verifies the signatures of the reputation data, which are considered correct if it is signed correctly from two different platforms, the agent itself and each reputation value by the agent which rated. If any error occurs during these tests registration fails.

After this step another platform has to verify the data a second time to save it within the DHT. For this purpose another platform responsible for the agent-identification data structure is chosen If there is no such platform, any other platform is chosen and the reputation data structure is sent to it in order to be verified.

On the other platform the Security Service once again checks the signatures . After correctly verifying the signatures the reputation data is passed on to the Data Service. The Data Service finally updates the reputation data in the DHT to the valid values. After that the agent, is successfully registered again.

Each agent has to renew its Lease before it expires. If a lease is not renewed in time the responsible Lease Service rates the agent with a poor reputation value. This reputation value is assigned to a special role called *System*. This role reflects the behavior of an agent in interaction with the reputation management system and is further described in section 4.8. When a lease is renewed correctly, it is reset in all Lease Services of the responsible platforms and a new lease containing the new timespan is returned to the agent.

Counter measures against attacks

An agent leaving the system with his reputation data structure can, in case the reputation

AA5	Change of identity

Table 3: Attacks: an Agent registers

is very bad, be tempted to come back with a new identifier (Attack AA5). Since neither the participating agents in the network nor the platform, the agents registers to, can prove the identity of this agent a counter measure for this attack outside the reputation management system is required. A new agent can be identified by any other agent as it has no reputation values assigned to it. So each agent can decide if it wants to cooperate with a new agent.

An agent can also keep its identifier and return without its reputation data. In this case the lease mechanism will ensure that the reputation data kept within the system will be of very bad value when the agent returns after a certain timespan. Therefore it will always be disadvantageous to come back with its old ID but not with its checked out reputation data structure.

In case the registering agent comes back with a reputation data structure belonging to another agent, even if it took over the other agent identifier, it would not be able to register because of the invalid signatures on the reputation data . If the agent also possesses the private key of the other agent there is no way to prevent the agent from registering correctly.

Finally the agent could try to leave the system once it has a very good reputation, save this reputation and return. After that it could behave bad, de-register and try to register again using the good reputation data saved before. To avoid this the up-to-date reputation data, handed out to each agent, is stored in a queue on the platforms responsible to manage data of this agent. If the agent registers again this reputation data can be compared with the data the agent provides. This way an agent can only register with its up-to-date reputation data structure.

4.7 An agent de-registers

In the following section de-registration of an agent from the reputation management system is illustrated. The system also allows agents to leave the system and come back with their accumulated reputation data.

Protocol flow
An agent de-registers from the plug-in, running on its platform, by providing its identifier.

The Reputation Service of this platform first retrieves the data structure describing the agent. For each role listed in this data structure the Reputation Service retrieves all reputation data saved in the chord DHT and creates a new data structure that the agent can save. This way any agent which has been participating in the network before can come back with its accumulated reputation data, related to the point in time on which it has left the network.

After assembling all the relevant reputation data for the leaving agent it gets signed with the platforms private key. A second platform to sign the data structure is a platform that is responsible for the data of the agent or if there is no such platform any other platform in the network. If no other platform can be found in the network the de-registration of the agent is considered to be failed.

Counter measures against attacks
With help of this protocol the same attacks as in section 4.6 are addressed.

4.8 An agent rates other agents

During the process of rating other agents the three essential concepts are used: a so-called *ticket*, a so-called *packet*, and a special role named *System Role*.

The *ticket* is a data structure, which all agents taking part at the transaction have to sign. It is issued by the reputation management system and also signed by all platforms that are responsible to manage data for the agent, that requested the ticket. A *ticket* is provided by reputation management on a per transaction basis. The *ticket* is signed by each participating agent. For this purpose an agent adds a time stamp (of its local time) to the *ticket* and signs it. The time stamp allows ordering of transactions for each agent without requiring a global time. The ordering is required for reputation management to decide which ratings are the most recent for each agent. The *ticket* testifies that all agents, which added a signature, agree to participate in the cooperation[4].

Closely related to a ticket is a data structure called *packet*, which is created on request of an agent by the reputation management serivce fter all relevant agents have signed a ticket. All

[4]This can be extended with details of the underlying joint commitment of the cooperation to be further integrated into the CPS process and provide more information about the context of the cooperation.

rating values (per rating agent, rated agent, and role of rated agent) applying to this transaction are stored in this data structure. In order to make ratings verifiable and non-manipulable every rating value is signed by the agent providing it.

The ratings contained within a packet are stored within the DHT in a data structure called *valuation*. There is a list of valuations for each agent and each role played by that agent. The *valuation* contains the identifier of the agent that provided the rating, the value of the rating, the ticket of the transaction associated with the rating, and signatures the platforms that were responsible to create the ticket and associated packet in order to testify the authenticity of the *valuation*.

Related to ticket and packet is a special role called *System Role*, whose valuations are set by the reputation management system. If an agent orders a packet a lease is created. If this lease expires before an agent returns a packet, the agent gets a poor rating for the *System Role*. Valuations can only be issued until the lease has expired. The valuation of an agent for the *System Role* is provided with every request of the reputation of an agent. Therefore other agents can conclude if an agent conforms to the rules of the reputation management. The use of the *System Role* encourages an agent to return a ticket and packet it has ordered before. An agent is also being rated in its *System Role*, if the lease expires that represents the presence of an agent within the system

If an agent wants to cooperate with and rate other agents, a ticket has to be ordered first. Then the ticket has to be signed by all agents taking part in the transaction to avoid faked transactions. After the transaction every agent involved can rate every other involved agent in one ore more particular roles by signing and adding a reputation value to the packet.

Protocol flow
According to section 3 all services beside the Consistency Service are involved in this protocol.

The agent that initializes the cooperation (*agent I* in the following) requests a new `Ticket` from its platform and states an expected duration of the intented cooperative work and its identifier. The ticket is created and signed by the platform, signed by another trusted platform, and then returned to the requesting agent. The expected duration and the identifier are stored within the ticket for later use.

Agent I passes around the ticket to all agents participating in the cooperative work. Each agent signs the ticket and thus signals readiness for cooperation. Additionally they add their local time stamp to get an unambiguous ordering of each agents ratings later.

After all relevant agents signed the ticket, *agent I* requests a `Packet` from its platform. The agent therefore passes the signed ticket to the platform, which checks if *agent I* is part of the reputation network and if the ticket is valid. If both checks return satisfactorily, the platform requests `ticket leases` from platforms that are also responsible for storage of data about *Agent I*, (called *platforms L* in the following) for the duration stored within the ticket. These platforms also verify the ticket. If this succeeds the lease is created on each *platform L* and the platform creates a signature for the `packet` to testify that it is valid. The packet is then returned to *agent I*.

After the cooperation has finished, *agent I* passes around the packet to all participating agents, which store their rating values for each other in the packet. Every agent has to sign its ratings to ensure traceability of who gave which rating to whom and data integrity.

After all relevant agents have stored their rating values, the packet can be provided to any platform. The platform then asks *platforms L*, whether the packet is still valid (the lease has not expired), which is determined by looking at the corresponding leases. If the packet is still valid each *platform L* provides a signature that testifies timely return of the packet. For this purpose not only the lease is checked but also each rating value is checked for a proper signature. If all *platforms L* testified with their signature that the packet is valid, the platform which received the packet creates a valuation data structure for each rating of an agent and stores it within the DHT. If the `ticket lease` expired, the whole `Packet` is ignored after *agent I* has already been punished.

A packet can be returned to any platform by any agent. Therefore a packet is valid until ratings for each combination of agents taking part at the cooperation has been stored in the DHT or the lease has expired. This is considered by the Lease Service of the *platforms L*.

Counter measures against attacks
Against the following attacks counter measures are taken here: Attack AA2 cannot be excluded

AA2	Faked transactions
AA3	Faked ratings
AA4	Harmful rating
AA6	Avoid rating by other agents

Table 4: Attacks: an Agent rates other agents

completely. A plug-in cannot make sure, that a transaction is really carried out. It is only possible to verify ratings by the means of checking the signatures of the ticket, the valuations and the distributed running ticket lease. To impair the unmeant effects of faked ratings (attack AA3) only one rating per agent pair and role is used to calculate the reputation value. So there are no effects to the complete system when two agents are trying to fake transactions.

Attack AA4 includes two cases: The first one - the packet is not signed by all agents involved - is completely excluded. If such a packet comes back to a platform, it is ignored, because not all agents rated in this structure have signed the ticket. The second case occurs if an agent signs a ticket and is rated harmful although it has done its job well. This problem is again out of the range of our plug-in, but is also impaired as in item AA3 above. All valuations to save have to be signed by the rating agent. If a malicious platform saves a not correctly signed valuation anyway, it is not considered when the reputation values are requested and calculated afterwards (Refer to the next section 4.9).

To avoid attack AA6, the cooperating agents themselves have to pay attention that all agents cooperating with them have signed the ticket. Otherwise they cannot give any ratings to them. So, the agents, which are not listed in the ticket, could possibly not cooperate without any personal (negative) consequences.

4.9 Reputation of an agent is retrieved

Another protocol which is covered in this subsection is the retrieval of agent reputation. The retriever has to provide the role and the agent identifier of the agent whose reputation has to be retrieved. As answer to this request the reputation of this agent playing the system role and the reputation of the platform that host the agent are also returned by the Reputation Service.

Protocol flow
First of all the request is forwarded to the Reputation Service. The Reputation Service in turn requests the valuations associated with the agent via the Data Service. These values are returned in historical order. Through a generic assignable calculating mechanism the Reputation Service calculates the actual value including the agent's values of the specified role and the system role. The newer ratings should usually be weighted higher than the older values.

Before answering the request, the Reputation Service has to compare the reputation of the platform with the marginal value (yellow card). Should the value be below this marginal value, the agent gets an alert. It is allowed to decide on its own, if it will work with a probably good agent on an evil platform.

Counter measures against attacks
All ratings are saved in priority queues, in which the most actual rating is at the beginning of

AA1	Pretending to be trustworthy
PA3	Fake of reputation data

Table 5: Attacks: Reputation of an agent is retrieved

the queue. To calculate the overall reputation value a formula like exponential smoothing (see also [PSEP01]) should be used, in order to weigh the actual ratings stronger.[Pad00] Thus this is a counter measure against attack AA1 because the effect of good valuations at the beginning of the participation in the network will fade out over time. To avoid the problem, created by attack PA3, all signatures of every rating are verified. So no other platform is allowed to fake ratings or create arbitrary rating values.

4.10 Checking consistency of data

This section describes how data consistency is checked in the plug-in. To achieve this the different internal steps necessary for a consistency check are explained. The consistency checks are an important function within the network in order to make sure that only properly working platforms are participating, which is crucial because all agents affiliated with the network rely on services provided by the platforms (e.g. the proposed plug-in). Since this plug-in is applied to open MAS the consistency checks exist to verify the accuracy and correctness of the saved data and measure the reliability of participating platforms.

Protocol flow
In general for this plug-in is assumed to control that the reliability of data has to be checked. Thus every platform acts for the agents hosted by it and triggers a consistency check after a certain number of events for the data concerning its agents.

Since it makes sense to conduct the consistency check only if there are at least two platforms in the network, the consistency checks in the network of one platform are disabled.

Agent Reputation Consistency Checks and *Platform Reputation Consistency Checks* verify the consistency of the distributed saved reputation data of agents and platforms. Platforms are considered to be a special type of agent.

Agent Data Consistency Checks and Platform Data Consistency Checks verify the consistency of the distributed saved identification data of agents and/or platforms. For any of the consistency checks the platform manager on a certain platform chooses a random agent identifier or its platform identifier to be verified and starts the test.

First of all the Platform Manager fetches the data from the chord DHT with help of the Data Manager. It After that the data is sorted on the original data and replicas. The original data is compared in order to determine a majority of correct data. The replica data is compared with the original only. Whether the original had a right data or not is not considered.

On the next stage the platforms with correct data are rewarded with a good reputation value and the platforms with incorrect data are punished with a bad reputation value. The reputation values are signed with help of the local Security Service and saved in the chord DHT with help of the local Data Service.

Counter measures against attacks
The whole consistency check concept is solely introduced to reveal platform behaviors like

PA1	Refusal of service
PA2	Reputation data manipulation
PA3	Fake of reputation data

Table 6: Attacks making Consistency Check necessary

the ones described in attacks PA1, PA2 and PA3. Whenever one of these is discovered the corresponding platform will get a bed rating and, if it continues acting incorrect, eventually be excluded from the network.

5 Implementation in Java

This section describes how the reputation management system, which is proposed in the preceding sections, is implemented in Java and with help of an existing chord implementation[5]

[5] `http://open-chord.sourceforge.net/`

which is also implemented in Java [KL06].

5.1 Interface of the plug-in

To provide only one interface to the agent platform and the agents hosted by it the facade pattern is used. It defines „a higher-level interface that makes the subsystem easier to use.“ [GHJV04] For users of the plug-in the whole complexity is hidden through this interface. Details of all protocols described in the preceding sections (e.g. request of a reputation value) are transparent to the agents and to the platform. This facilitates change of the internal implementation of reputation management without changing the applications using it.

This section contains a description of the interface used in our implementation: `de.uniba.wiai.lspi.trust.trustservice.TrustManager`. The methods of this interface can be divided into two categories: methods for platforms and methods for agents.

Methods for platforms
In order to join the open MAS, a platform has to provide its public key when calling the method `platformLogin`. A specific number of trusted platforms decides if the platform is accepted or not by signing its public key. To leave the network regularly, a platform has to call the method `platformLogout`. This method ensures that all responsibilities are rearranged within the reputation management system.

Methods for agents
The method `agentLogin` is provided with two different signatures. The first one, which contains no parameters is for new agents, which are registering with the network for the first time. The second one is the login for returning agents with a list containing signed reputations. In case an agent leaving the platform the `agentLogout` method provides the possibility to perform a logout. A list of double-signed reputation values is returned. The agent has to store its reputation, if it wants to provide it when registering again.

The method `getAgentRep` provides the possibility to agents to request a reputation value of another agent in a specified role. The returned data also contains informations about the reputation values of the platform the agent is running on and the reputation value of the agent regarding the *System Role*. Using the concept of leases for agents, there has to be a possibility to renew them. This can be performed by the method `renewAgentLease`.

According to section 4.8 methods to order a ticket (`newTicket`), to sign a ticket to signalize that a agent wants to take part in a transaction (`addSignature`) and to order a new packet (`newPacket`) are required. After a transaction has finished the agent provides the Packet with the ticket and the signed reputation values by method `setAgentRep` back to the plug-in.

5.2 Implementing the architecture with Java

This subsection describes how the architecture described in section 3 is implemented in the prototypic implementation and which patterns are used. First however, the general structure of packages will be described. For each service described in section 3 a package with a corresponding implementation of the service exists. Each such package consists of at least one interface which provides the methods to access the service implementation. The packages `dataservice`[6], `reputationservice` and `securityservice` encompass more than one interface each, because they provide individualized interfaces to some of the other services. Furthermore, each package uses a kind of static factory method provided by an additional class combined with a kind of singleton to ensure that each service can be instantiated only once. This method has been implemented in an additional class, which ensures, that there is only a single instance implementing the interface, that provides access to a service.

5.2.1 Layer 3

On layer three there is only the package `trustservice`. It contains only the interface explained in section 5.1 and its implementation, which forwards the request to the services concerned with the desired functionality. Which services are concerned depends on the method that is called. Like suggested in [GHJV04], a mediator pattern is used in all these cases. According to Gamma et al. the mediator pattern „defines an object that encapsulates how a set of objects interact.“ [GHJV04]. In out implementation the steering object is the service, which has the main job to do concerning a method. For example if „An agent rates another agent“ (section 4.8) or „Reputation of an agent is retrieved“ (section 4.9) the Reputation Service manages all actions and acts as the mediator.

5.2.2 Layer 2

To ensure that in each plug-in only one instance of each service is created, a kind of singleton pattern has been used as described above. If this instance is not existing jet, it is instantiated now. Instantiation of the Trust Service is performed by the agent platform using the plug-in, whereas the other services are initialized the first time they are requested by another service. By this pattern it can be ensured that all services of one plug-in are e.g. using the same Data Service. As a consequence using this static factories, only one plug-in can be instantiated in a single Java Virtual Machine (JVM).

For many use cases the `Reputation Service` is the mediator of all actions occurring while answering the request. Another pattern, which has been used here, is the Strategy Pattern. This behavioral pattern is used to make algorithms interchangeable. [GHJV04]

In the reputation management service the pattern is used to make the reputation metric and representation of reputation values exchangeable. First of all the metrics used for reputation and ratings is declared as soon as the Trust Service is instantiated. When a platform joins a

[6]The prefix `de.uniba.wiai.lspi.trust` is cut of from all package names for simplicity

MAS it must be checked that it uses the same strategy as the platforms already part of the network do.

Figure 4 presents the strategy implemented in our reputation management system. `ReputationContext` in this figure represents any service that must use the reputation values or metric. The strategy itself is represented by the class `ValueTypeInformation` and consists of the metric (`Rating`)and reputation value representation `ValueType`. An instance of this class containing the concrete `ValueType` and `Rating` is passed to the Trust Service when it is created. In this prototype two different ValueTypes are implemented: A so-called `FloatValueType`, which can contain float values between zero and one, and the `ValueType Grade`, which demonstrates the German school grade system between one (best grade) and six, whereas only integers are allowed. Through the generic architecture of our system, new `ValueType` implementation can be added, which extend `de.uniba.wiai.lspi.ss05.pi3.trust.model.ValueType`.

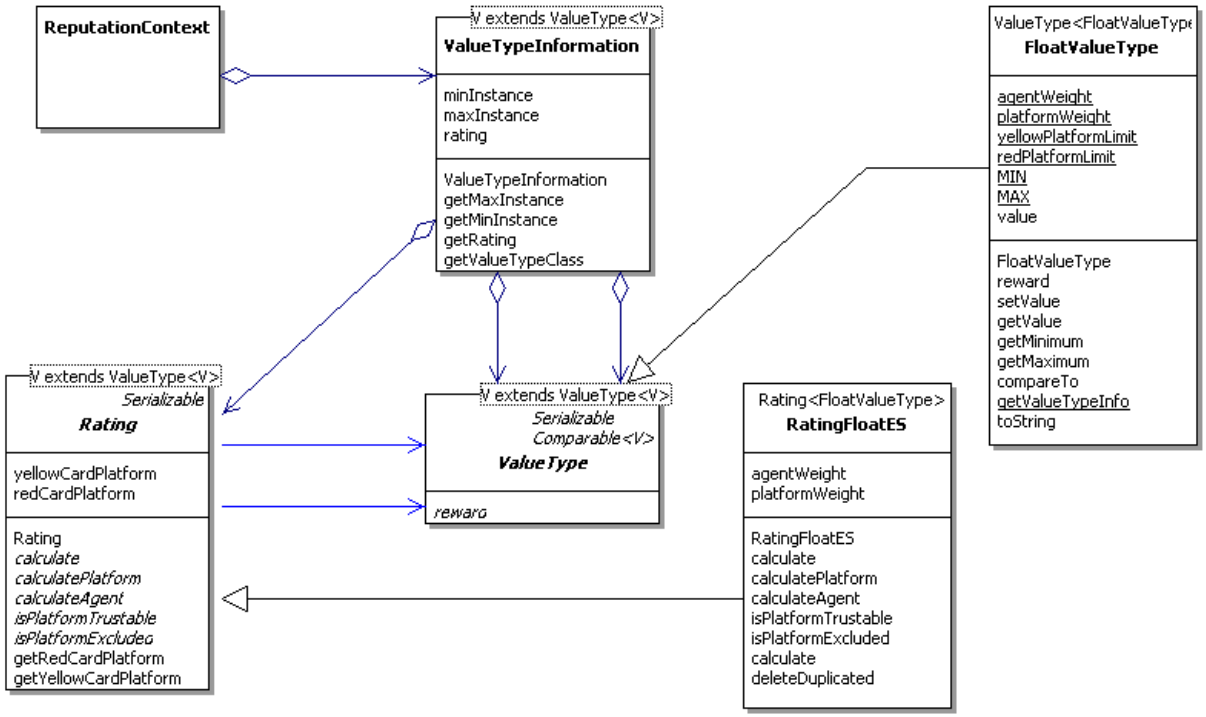


Figure 4: The metric strategy pattern

The strategy consists of a concrete `ValueType` and a metric `Rating` that can compute reputation values from this `ValueType`. Our prototype provides two alternative metrics, which can be used with `FloatValueType`. The first one, a very simple one, is the unweighted average. All ratings, which are requested from the Data Service, are just added and after that divided through their quantity. The alternative, called `RatingFloatES`, is more recommendable (compare to [PSEP01]). It is based on exponential smoothing. So newer ratings are weighted more and the protection against attack AA1 is much better than using the other alternative. Among other things the configuration of the values for the exponential smoothing (like weights) depends on the system size. They can be denoted in a configuration file and are independent from the values of other platforms in the system.

Main tasks of the *Platform Service* are to manage the join procedure of a platform and to

ensure the consistency of the distributed data in the chord DHT. These `Consistency Checks` proceed after a certain time span and check the data of agents running on the local platform. `Consistency Checks` are performed as described in 4.10. To speed up the checking instead of comparing each value the mechanisms provided by the `Reputation Service` is used to calculate a actual reputation value which is compared afterwards.

Managing agent and ticket leases is the main task of the *Lease Service*. To avoid attacks based on cooperation between a platform and an agent running on that platform, agent leases are not managed on the platform hosting that agent. They are stored on the platforms responsible to host the identification data of that agent. Also ticket leases are distributed in this fashion and for the same reason. Leases are stored in special local data structures which are periodically checked by a thread. When one of the leases on this platform expires, the agent to that the lease belongs is rated harmful with help of the Reputation Service. If the request to extend a lease or extend the deadline for rating of a transaction associated with a ticket arrives in time, the dataset will be removed so that the lease cannot expire any more. The different platforms responsible for the lease do not coordinate with each other to ensure that a lease really expired. This is not necessary as any inconsistencies that may arise are detected by the Platform Service and can be repaired later on.

All data and communication has to be verified with help of the *Security Service*. In general the *Security Service* is implemented based on the packages `java.security` and `javax.crypto`. For our reputation management system an asymmetrical ciphering algorithm for the key pairs of every participant and a symmetrical ciphering algorithm to be able to provide the Pretty Good Privacy (PGP) Protocol [Zim95] for communication between platforms is required.

To create the key pairs (`java.security.KeyPair`) for every participant in the network (agents and platforms) `java.security.KeyPairGenerator` is applied with a asymmetrical algorithm, that is set to the same value for the whole MAS. This must also be ensured when a platform joins a MAS. Currently RSA is used for that purpose. To create the symmetrical keys used for the PGP Protocol the `javax.crypto.KeyGenerator` with the DESede (Triple DES) algorithm was used. All the keys are generated by an instance of the class called `KeyGenerator` located in the `securityservice` package. All sign-able data structure classes are derived from the class `TrustStruct`. To add a signature to such a `TrustStruct` the following steps are proceeded:

First the data structure to be signed is converted into a byte array. Then `java.security.-Signature` class is used to create a byte array that represents the signature for the `TrustStruct`. This is created using the private key, given as an argument, from the participant signing the `TrustStruct`, and a signature algorithm (to be able to use the created RSA keys `SHA1WITHRSA` is used). Finally after creating the signature for the `TrustStruct` a new `SignedTrustStruct` instance is created containing the signed `TrustStruct`, the signers identifier, the signature and a flag indicating if or if not the `SignedTrustStruct` may be signed again. Every additional signature created for this data structure is created using the same mechanisms. Since there can be a need to distinguish between signature of an agent a platform, `SignedTrustStruct` contains two data structure to store either signatures of agents or platforms. Instances of `TrustStruct` and their signatures are encapsulated in instances of `SignedTrustStruct` to make it possible to distinguish between the instance without signature and with signature.

To verify a signature again java.security.Signature is used. Providing the signed object (extracted from a `SignedTrustStruct`) (converted into a byte array) the byte array representing the signature and the corresponding public key the signature is verified. The `SecurityManagerImpl` object acts as a mediator for this action: it converts the signed object, using the `SignatureUtil` object, and retrieves the public key for the participant of the network which signed the object. All conversion, signing and verification is carried out in the `SignatureUtil` object also within the `securityserivce` package.

The *Security Service* also provides facilities to secure communication between agent platforms based on PGP used by the Communication Service on layer 1.

Each message of a message session is enciphered with help of a symmetric session key. In order to provide the symmetric session key to the remote communication partner it has to be enciphered with help of the communication partner's public key. To achieve this the secret session key is asymmetrically enciphered with the other party's public key with help of `javax.crypto.Cipher` is used. By providing the public key of the recipient the secret key is enciphered asymmetrically and returned as a byte array. To decrypt a encrypted secret key on the recipient platform `javax.crypto.Cipher` is used again. All these actions have been encapsulated in a class called `CipherUtil` within the `securityservice` package, while the `SecurityManager` object again acts as a mediator.

In order to symmetrically encrypt the actual message objects the message is converted into a byte array and then again enciphered by `javax.crypto.Cipher` (using the symmetrical algorithm and the secret session key now). Analogously the byte array representing the encoded message object is decoded, using `javax.crypto.Cipher` and the secret key, on the platform receiving the message.

5.2.3 Layer 1

Although there already exists a communication network for the chord DHT, the reputation management service has to provide the communication between interacting platforms. Therefore the `Communication Service` offers methods to contact remote platforms for specific tasks of the other Services. Throughout there is need to send remote method calls to other platforms e.g. to request a signature from that platform. For this purpose a service invokes the corresponding method of the Communication Service. On the remote platform the local Communication Service delegates the call to the responsible services.

For the communication between platforms the PGP Protocol has been chosen. According to that protocol messages are enciphered with a symmetrical session key known to both parties of the communication session. To initiate such a communication and therefore publish the secret session key to the other party the first message of such a session will consist of the following parts: the message object, enciphered using the secret session key and the secret session key enciphered with the other party's public key. This way the other party can decipher the session key and thus decode the whole message. And furthermore the session key is now known to both parties of the communication session. The means to en-/decipher messages are provided by the Security Service.

The *Data Service* in a class called `DataManager` which is responsible for saving data within the chord implementation or to retrieve data from it. How this is done, is transparent to the layers above. Within chord data replication is already implemented to ensure that no data is lost when a peer crashes. However, this kind of replication only means, that data from one peer is stored on a number of succeeding peers. This type of replication is not sufficient to avoid, that a peer which is responsible for storing data, is unnoticed changing data, because there is no reference data to compare the stored data with. Therefore, in addition to the replication already implemented in chord, the *Data Service* stores each data item on more than one peer. This yields in several advantages:

- By retrieving data several times from chord and comparing these items, the `DataManager` is able to recognize if a platform has changed data or removed data.
- After recognition of data change the Data Service saves his findings in a queue which can be accessed by other services (e.g. the Platform Service). Thus, the `DataManager` can report malfunctioning platforms indirectly.
- The `DataManager` can prevent provision of inconsistent data to higher layers. It can do so by retrieving the requested data several times from chord, performing a majority check and returning the data which was found to be correct.

For each data item the `DataManager` wants to store within chord, it has to create a unique identifier. To store data several times within chord, the `DataManager` simply creates a (configurable) number of identifiers for a single item and stores the item several times, associated with each identifier. How many identifiers are created for each item, i.e. how often each item is stored within Chord, has to be specified while initializing the `DataManager`. This value is globally set for all platforms and their `DataManager` within the reputation management system.

The majority check mentioned above is implemented with help of a private method and tries to determine from a list of data those data that represent a majority. This private method is called from every method that retrieves data from the chord DHT. The number of data items that represent an absolute majority is calculated first. Then the data items are compared with each other until an absolute majority has been found. One data item from this majority is then returned. If there is no absolute majority found, the data represented by a simple majority is returned. This indicates however, that there are inconsistencies within the chord DHT. This information is stored using a flag, which can be read by other Managers (e.g. the platform manager to check for inconsistencies).

Since data items are stored several times within chord, the `DataManager` has to ensure, that the stored data is consistent. This has to be done especially to prevent that an agent platform is punished for storing incorrect data. There are two things, the `DataManager` has to do. First, it must prevent that wrong data is stored on its own platform or that data is missing on its own platform. Secondly, the `DataManager` must provide the already mentioned means to find inconsistencies within chord, thus enabling the punishment of the responsible agent platforms.

Some operations which are provided by the `DataManager` are context sensitive, i.e. in some cases they may be performed, in other cases, they may be not, depending on the type of data

that would be affected by the operation. These operations are insertion, deletion and update of data. If an operation may be performed is not checked by the `DataManager`, but within chord. However, there had to be some additional `DataManager` methods to support those checks and the chord implementation had to be extended with security features.

Since the `DataManager` stores data several times within chord and for every request all data items have to be retrieved to perform a majority check, data requests are expensive. Therefore, it is reasonable to cache data, which do not change anymore. In the current implementation this data is stored in a local map within the `DataManager` and not removed anymore. In case the plug-in is developed any further, it surely is sensible to remove data from this map using a scheme that has still to be defined. One could e.g. remove the oldest entries after a certain time, or one could remove those entries, which are used seldom.

5.2.4 Layer 0

Communication of the plug-in is based on two technologies. On one side agent platforms have to communicate directly and on the other side they use a DHT -namely chord- to store data in a P2P network. The DHT itself again uses Java sockets to facilitate communication between single agent platforms. This section first describes how direct communication between agent platforms is implemented using Java Sockets. Afterwards it is explained how an already existing implementation of a chord overlay network uses Java Sockets and is extended to fit the requirements for application in an environment where non-trustworthy peers may be incorporated.

Socket communication The provided send-methods of the Communication Service Interface facilitate remote method calls. Before a message is sent to another platform there is a check if the receiver address is the own address of the platform. This can happen when e.g. the *Lease Service* has to send a message to all responsible platforms of a specific agent for whom it is responsible too. Then there is no remote call over the network but a local call to the responsible Service of the local platform. When the receiver address and the local platform address are different the message has to be sent over the network. Therefore the message is enciphered, signed, and with the so-called secret key packed into an envelope. The secret key represents the session key for one interaction session of two platforms. An envelope is a data structure that contains the sender address, the receiver address, the type of the message, the encrypted message and finally the secret key encrypted with help of the receivers public key. The next step is to open a new socket to the receiver platform and to send the envelope. Now the socket waits for a response from the receiver platform. Thus, the communication is synchronous for the service that calls the method. The incoming result message can contain an object or an exception. The object must be of the type of the method that has been called and serves as return value. If it is not an exception is raised. The exception -if present- represents an exception that occurred on the remote platform. So that the socket has not to wait infinitely, a timeout is set. If the timeout elapses an exception is thrown. All send methods operate this way.

To receive messages the *Communication Service* makes use of a multi-threaded server pattern The class `ServerSocketListener` provides a server socket, that enables the plug-in to listen

for incoming messages which have to be of the type envelope. Incoming envelopes are checked on the message type they contain based on which the right method can be called by the *Communication Service*.

A mechanism to avoid denial-of-service attacks is implemented in the socket layer. To achieve this a remote platform is only allowed to send a configurable number of requests in a certain (also configurable) period of time. Finally a malicious platform, that exceeds the number of requests is added to a blocked platform list. From platforms on that list no incoming requests are accepted anymore. After a globally set period of time a platform is removed from the list of blocked platforms. This mechanism guarantees the performance of the plug-in because without it a malicious platform would be able to send thousands of envelopes with messages which have to be decrypted. Such an attack would probably slow down the plug-in enormously since the decryption process is expensive.

When an incoming message is received, the `ServerSocketListener` checks if the requesting platform is allowed to send an envelope at all. Then the `ServerSocketListener` updates the number of requests received from that platform. Assumed the platform is still allowed to send requests, the `ServerSocketListener` instantiates a class called `RequestHandler`, which has its own thread of control and is provided with the socket representing the just opened connection. The `RequestHandler` expects an envelope as incoming message on its socket. Hence, if the incoming object is not an envelope the request is ignored, the connection is closed and the thread is terminated. If the incoming object is an envelope, a second security check is applied, that checks if the requesting platform is trustable and on that score if the connection is allowed.

Chord implementation An already existing open source Java implementation of the chord overlay network is adopted to serve as DHT for the implementation of the reputation management system. This implementation (called Open Chord[7] [KL06]) is based on Java sockets and provides an interface to Java Applications to use the DHT functionality. Synchronous and Asynchronous methods to lookup, insert and delete data from the DHT are provided. The implementation transparently performs the maintenance of the chord DHT as described in section 4.1. It also transparently replicates data stored within the DHT by copying it to a configurable number of successors of a peer. Figure 5 presents a high level overview of the Open Chord architecture.

.

The interfaces `AsynChord` and `Chord` provide methods to a Java application (here: our reputation management system) to lookup a peer responsible for data with a certain identifier and to insert, delete, update, and retrieve data. `AsynChord` allows to perform these operations in an asynchronous fashion while `Chord` provides methods to synchronously perform these operations.

The communication abstraction layer provides two abstract classes: `Proxy` and `Endpoint`. `Proxy` represents references to remote peers that a peer holds in its finger table and provides methods that can be invoked on these remote peers. `Endpoint` represents the instance that allows a peer to receive incoming method calls from remote peers.

[7] `http://open-chord.sourceforge.net/`

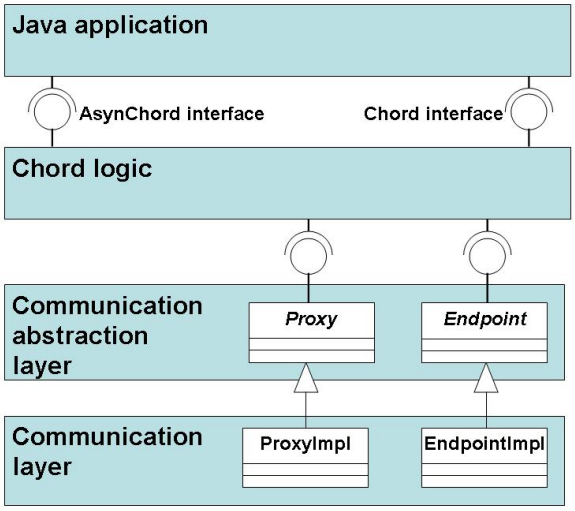


Figure 5: Architectural overview of Open Chord.

The communication layer is implemented with help of subclasses of `Proxy` and `Endpoint`. `Proxy` represents a peer at a remote peer and facilitates remote method invocation to the represented peer. `Endpoint` represents an instance that listens for incoming method calls for a peer.

For communication over sockets each peer is represented at other peers with help of a so called `SocketProxy`. This proxy sends requests for method invocations to the `SocketEndpoint` of the peer it represents. For this purpose `SocketEndpoint` is listening for incoming requests of other peers with help of a server port and acts as a multi-threaded server. For each method (e.g. retrieve/insert an entry from/into the DHT) a message is send over a socket to the remote `SocketEndpoint` by a `SocketProxy`.

As this implementation of chord assumes that all participating peers are benevolent and trustworthy some modifications and security enhancements have to be made. These add-ons and changes are described in the following paragraphs.

TrustSecureAccess The socket communication layer of Open Chord has been extended to use an instance of a security manager. Each `SocketProxy` and `SocketEndpoint` holds a reference to the security manager of the local peer. This security manager on the one side checks if an outgoing method request with help of a `SocketProxy` is permitted. On the other side `SocketEndpoint` uses it to check if incoming requests are valid and the requested method may be performed by the requesting remote peer. The security manager can also be used to verify, sign, and en-/decipher messages send by a `SocketProxy`. In section 5.2.3 has been mentioned, that for the operations insert, delete and update, checks have to be made whether these operations are permitted. The security manager provides a method that checks permission for each method that can be called from a remote peer provided by `Proxy` respectively `SocketProxy`.

The security manager instance must be set before a chord overlay network is joined by a peer. If no security manager is set a standard security manager which permits all outgoing and incoming methods to be performed without being verified etc. is used.

`TrustSecureAccess` is the security manager implementation provided to be used with our reputation management system. It holds references on the local `DataManager`, the local *Security Service* and the local *Reputation Service* to perform its tasks. To ensure that critical messages (e.g. messages that request the removal of a data set) are only considered if they were sent by registered, well behaving trustworthy agent platforms, these messages have to be signed and verified. The *Security Service* provides all crypthography related methods to accomplish this. As mentioned in section 5.2.3 there are some constraints for the execution of the operations insert, delete and update. Compliance to these constraints is ensured within `TrustSecureAccess`. It is checked for what kind of data an operation should be executed and if this execution is allowed. This decision is based on the trustworthiness of the remote peer and the affected data. For this purpose the local *Reputation Service* is used to determine if remote method calls with help of `SocketProxy` and incoming method calls via `SocketEndpoint` are made to/by a trustworthy remote peer.

The reference to `DataManager` is required as the identification data of remote peers has to be retrieved from the chord overlay network itself and `DataManager` provides these methods. Therefore the communication within the chord overlay network requires access to the chord overlay network itself. Because of this fact special care has to be taken to avoid deadlocks. This required that the join process of a peer must be changed.

In the existing chord implementation joining a chord overlay network and creation of an identifier for a peer is based on the network address of a peer. This had to be changed in order to prevent that an agent platform can join the overlay network at a certain point in the chord ring by choosing a unique identifier on its own and to avoid deadlocks during the join process.

The identifier of a peer determines where it will be located within the chord ring. The identifier is generated from a data set which contains the identifier of the agent platform it represents, the public key of that platform and the address of the peer. To join a network a platform first contacts a platform already part of the MAS with help of its local Communication Service. From this platform it receives the permission to join the network. To enable the join this platform inserts the identification data structure of the new platform into the chord DHT. The identification data first has to be signed by another platform. This platform can check if the data provided is valid and decline the join of the new platform. Since, the new peer has no means to influence the signature of the other platform and since the signature is included into the identifier generating process, the new peer has no control over its identifier and thus no control over where it will be located within the chord ring structure. Before joining the MAS the platform provides the assigned identifier for the chord network to its local chord interface. After its identifier being assigned the new peer has to contact its successor to take over the responsibility for data as described in section 4.1. The successor with help of `TrustSecureAccess` tests whether the joining peer is taking over responsibility for the right data with help of the signed identification data which has been inserted into the DHT. Only if approved the new node will be able to join the chord overlay network.

Peers or respectively the agent platforms they represent can become non-trustworthy. Therefore not only the join process had to be modified, but also peers must sometimes be excluded from the network. For this purpose the maintenance tasks of peers are exploited. During these tasks the availability of the predecessor and successor of a peer are checked with help of a

ping message. If one peer wants to ping another peer, it determines with help of `TrustSecureAccess` if this is permitted. Based on reputation of the remote peer `TrustSecureAccess` then decides if it is still trustworthy. To make this decision `TrustSecureAccess` has to ask the local *Reputation Service* if the other peer is still trustworthy. If a peer is determined to be trustworthy by the *Reputation Service* it is stored in a local list within `TrustSecureAccess` for a while. Thus, `TrustSecureAccess` does not have to use the *Reputation Service* each time it needs the reputation of another peer. Similarly, non-trustworthy peers are saved in a list. If the remote peer is non-trustworthy (regardless whether successor or predecessor) it is disposed from the finger table of the local peer.

If a the endpoint of a peer receives a ping from a remote peer `TrustSecureAccess` also checks if that node is still trustworthy. To avoid deadlocks (that can occur as the local peer requires access to the chord overlay network to check reputation of another peer), the ping is answered immediately, except if the node is already noted in the list of non-trustworthy peers. If the ping was answered immediately the reputation check is performed asynchronously with help of a thread and the ping is discarded the next time if the remote peer is non-trustworthy.

By not communicating with a non-trustworthy peer and removing references to it from the local finger tables of trustworthy peers it is excluded from the chord overlay network, as it requires its successor and predecessor relationships.

6 Conclusion

This paper presented a fully decentralized scheme for reputation management in MAS based on P2P technology. It provides protocols to store, update, and retrieve reputation data for agents and platforms and to counter measure attacks on reputation management. The application of our reputation management scheme in many domains is supported by its implementation in Java, its modular design, and the possibility to exchange the employed reputation metric and reputation value representation.

Mechanisms to identify and authenticate agents and platforms have been developed and it has been shown how an existing implementation of a structured P2P overlay network (namely chord and its Java implementation Open Chord) can be extended and employed to support distributed reputation management. The mechanisms of our reputation management system allow to rate agents and their behavior in CPS and behavior of agent platforms regarding provision of the distributed reputation management service to create reputations for agents and agent platforms. These mechanisms to identify and authenticate agents and platforms should be extended to rely on the owner of an agent or a platform, so that more sophisticated methods to prevent attacks on reputation management are possible. Knowledge about the owner of an agent or a platform adds extra possibilities to avoid faked ratings and transactions. It also facilitates tracking of agent and platform owner behavior. For this purpose the data structures to identify agents and platforms can be extended to also contain the identity of their owners. The owners should have an identity certified by a trusted third party outside the MAS. To keep track of an owner's reputation it would be possible to treat it as a special agent with the special role owner, that then can be used with the existing mechanisms presented in this paper. Then, it would be

possible to prevent new platforms or agents of owners with low reputation to enter the MAS, and this would increase the overall trust in the MAS.

To better integrate CPS and reputation management and to provide more information on the context of a cooperation, the joint commitment underlying a cooperation should be integrated into the data structures used during the process of rating agents and into the data structures representing the reputation of an agent.

To further develop our approach it will be evaluated in different application domains and with different reputation metrics. An additional step is the extension by services supporting the ontological dimension (described by [SS02b]) by providing knowledge about roles and their relationships. These services may be used by all agents to store their experience about the relationships that exist between roles. This can facilitate the estimation of trust in an agent in a role based on the relations to other roles.

References

[ARH00] Alfarez Abdul-Rahman and Stephen Hailes. Supporting Trust in Virtual Communities. In *HICSS '00: Proceedings of the 33rd Hawaii International Conference on System Sciences-Volume 6*, page 6007, Washington, DC, USA, 2000. IEEE Computer Society.

[CBG02] Jonathan Carter, Elijah Bitting, and Ali A. Ghorbani. Reputation Formalization within Information Sharing Multiagent Architectures. *Computational Intelligence*, 18(4):45–64, 11 2002.

[CF98] C. Castelfranchi and R. Falcone. Principles of Trust for MAS: Cognitive Anatomy, Social Importance, and Quantification. In *ICMAS '98: Proceedings of the 3rd International Conference on Multi Agent Systems*, page 72, Washington, DC, USA, 1998. IEEE Computer Society.

[CP02] Rosaria Conte and Mario Paolucci. *Reputation in arificial Societies*. Kluwer Academic Publisher, 2002.

[CSG+03] V. Cahill, B. Shand, E. Gray, C. Bryce, and N. Dimmock. Using trust for secure collaboration in uncertain environments. *IEEE Pervasive Computing*, 2:52–61, 2003.

[DKV03] Barbara Dunin-Keplicz and Rineke Verbrugge. Dialogue in Teamwork, 2003.

[FIP02] FIPA TC Architecture. *FIPA Abstract Architecture Specification*. FOUNDATION FOR INTELLIGENT PHYSICAL AGENTS, 12 2002.

[GH04] Jennifer Golbeck and James Hendler. Accuracy of Metrics for Inferring Trust and Reputation in Semantic Web-Based Social Networks. In *Proceedings of 14th International Conference on Knowledge Engineering and Knowledge Management*, 2004.

[GHJV04] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 2004.

[GLd02] Nathan Griffiths, Michael Luck, and Mark d'Inverno. Annotating Cooperative Plans with Trusted Agents. In *Trust, Reputation, and Security*, pages 87–107, 2002.

[JSW98] Nicholas R. Jennings, Katia Sycara, and Michael Wooldridge. A Roadmap of Agent Research and Development. *Autonomous Agents and Multi-Agent Systems*, 1(1):7–38, 1998.

[KL06] Sven Kaffille and Karsten Loesing. *Open Chord version 1.0 - UserŠs Manual*. Lehrstuhl für Praktische Informatik, Fakultät WIAI, Otto Friedrich Universität Bamberg, FeldkirchenstraSSe 21, D-96047 Bamberg, Germany, 2006.

[Pad00] Boris Padovan. *Ein Vertrauens- und Reputationsmodell für Multi-Agenten Systeme*. PhD thesis, Albert-Ludwigs-Universität Freiburg im Breisgau, 2000.

[PSEP01] Boris Padovan, Stefan Sackmann, Thorsten Eymann, and Ingo Pippow. Automatisierte Reputationsverfolgung auf einem agentenbasierten elektronischen Marktplatz. In Hans-Ulrich Buhl, Andreas Huther, and Bernd Reitwiesner, editors, *Internationale Tagung Wirtschaftsinformatik 2001*, pages 517–530, 2001.

[PV02] Ioannis Partsakoulakis and George Vouros. Importance and Properties of Roles in MAS Organization: A review of methodologies and systems. In *Proceedings of the workshop on MAS Problem Spaces and Their Implications to Achieving Globally Coherent Behavior*, 2002.

[SMK+01] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications*, pages 149–160. ACM Press, 2001.

[SS02a] Jordi Sabater and Carles Sierra. Reputation and social network analysis in multiagent systems. In *AAMAS '02: Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, pages 475–482, New York, NY, USA, 2002. ACM Press.

[SS02b] Jordi Sabater and Carles Sierra. Social ReGreT, a reputation model based on social relations. *SIGecom Exch.*, 3(1):44–56, 2002.

[WJ99] M. J. Wooldridge and N. R. Jennings. Cooperative Problem Solving. *Journal of Logic and Computation*, 9(4):563–592, 1999.

[WS00] H. Chi Wong and Katia P. Sycara. Adding Security and Trust to Multiagent Systems. *Applied Artificial Intelligence*, 14(9):927–941, 2000.

[Zim95] Phil Zimmermann. *The Official PGP User's Guide*. The MIT Press, 1995.

[ZM00] Giorgos Zacharia and Pattie Maes. Trust Management Through Reputation Mechanisms. *Applied Artificial Intelligence*, 14(9):881–907, 2000.

A List of previous University of Bamberg reports

Bamberger Beiträge zur Wirtschaftsinformatik

Stand April 3, 2006

Nr. 1 (1989) Augsburger W., Bartmann D., Sinz E.J.: Das Bamberger Modell: Der Diplom-Studiengang Wirtschaftsinformatik an der Universität Bamberg (Nachdruck Dez. 1990)

Nr. 2 (1990) Esswein W.: Definition, Implementierung und Einsatz einer kompatiblen Datenbankschnittstelle für PROLOG

Nr. 3 (1990) Augsburger W., Rieder H., Schwab J.: Endbenutzerorientierte Informationsgewinnung aus numerischen Daten am Beispiel von Unternehmenskennzahlen

Nr. 4 (1990) Ferstl O.K., Sinz E.J.: Objektmodellierung betrieblicher Informationsmodelle im Semantischen Objektmodell (SOM) (Nachdruck Nov. 1990)

Nr. 5 (1990) Ferstl O.K., Sinz E.J.: Ein Vorgehensmodell zur Objektmodellierung betrieblicher Informationssysteme im Semantischen Objektmodell (SOM)

Nr. 6 (1991) Augsburger W., Rieder H., Schwab J.: Systemtheoretische Repräsentation von Strukturen und Bewertungsfunktionen über zeitabhängigen betrieblichen numerischen Daten

Nr. 7 (1991) Augsburger W., Rieder H., Schwab J.: Wissensbasiertes, inhaltsorientiertes Retrieval statistischer Daten mit EISREVU / Ein Verarbeitungsmodell für eine modulare Bewertung von Kennzahlenwerten für den Endanwender

Nr. 8 (1991) Schwab J.: Ein computergestütztes Modellierungssystem zur Kennzahlenbewertung

Nr. 9 (1992) Gross H.-P.: Eine semantiktreue Transformation vom Entity-Relationship-Modell in das Strukturierte Entity-Relationship-Modell

Nr. 10 (1992) Sinz E.J.: Datenmodellierung im Strukturierten Entity-Relationship-Modell (SERM)

Nr. 11 (1992) Ferstl O.K., Sinz E. J.: Glossar zum Begriffsystem des Semantischen Objektmodells

Nr. 12 (1992) Sinz E. J., Popp K.M.: Zur Ableitung der Grobstruktur des konzeptuellen Schemas aus dem Modell der betrieblichen Diskurswelt

Nr. 13 (1992) Esswein W., Locarek H.: Objektorientierte Programmierung mit dem Objekt-Rollenmodell

Nr. 14 (1992) Esswein W.: Das Rollenmodell der Organsiation: Die Berücksichtigung aufbauorganisatorische Regelungen in Unternehmensmodellen

Nr. 15 (1992) Schwab H. J.: EISREVU-Modellierungssystem. Benutzerhandbuch

Nr. 16 (1992) Schwab K.: Die Implementierung eines relationalen DBMS nach dem Client/Server-Prinzip

Nr. 17 (1993) Schwab K.: Konzeption, Entwicklung und Implementierung eines computergestützten Bürovorgangssystems zur Modellierung von Vorgangsklassen und Abwicklung und Überwachung von Vorgängen. Dissertation

Nr. 18 (1993) Ferstl O.K., Sinz E.J.: Der Modellierungsansatz des Semantischen Objektmodells

Nr. 19 (1994) Ferstl O.K., Sinz E.J., Amberg M., Hagemann U., Malischewski C.: Tool-Based Business Process Modeling Using the SOM Approach

Nr. 20 (1994) Ferstl O.K., Sinz E.J.: From Business Process Modeling to the Specification of Distributed Business Application Systems - An Object-Oriented Approach -. 1st edition, June 1994

Ferstl O.K., Sinz E.J. : Multi-Layered Development of Business Process Models and Distributed Business Application Systems - An Object-Oriented Approach -. 2nd edition, November 1994

Nr. 21 (1994) Ferstl O.K., Sinz E.J.: Der Ansatz des Semantischen Objektmodells zur Modellierung von Geschäftsprozessen

Nr. 22 (1994) Augsburger W., Schwab K.: Using Formalism and Semi-Formal Constructs for Modeling Information Systems

Nr. 23 (1994) Ferstl O.K., Hagemann U.: Simulation hierarischer objekt- und transaktionsorientierter Modelle

Nr. 24 (1994) Sinz E.J.: Das Informationssystem der Universität als Instrument zur zielgerichteten Lenkung von Universitätsprozessen

Nr. 25 (1994) Wittke M., Mekinic, G.: Kooperierende Informationsräume. Ein Ansatz für verteilte Führungsinformationssysteme

Nr. 26 (1995) Ferstl O.K., Sinz E.J.: Re-Engineering von Geschäftsprozessen auf der Grundlage des SOM-Ansatzes

Nr. 27 (1995) Ferstl, O.K., Mannmeusel, Th.: Dezentrale Produktionslenkung. Erscheint in CIM-Management 3/1995

Nr. 28 (1995) Ludwig, H., Schwab, K.: Integrating cooperation systems: an event-based approach

Nr. 30 (1995) Augsburger W., Ludwig H., Schwab K.: Koordinationsmethoden und -werkzeuge bei der computergestützten kooperativen Arbeit

Nr. 31 (1995) Ferstl O.K., Mannmeusel T.: Gestaltung industrieller Geschäftsprozesse

Nr. 32 (1995) Gunzenhäuser R., Duske A., Ferstl O.K., Ludwig H., Mekinic G., Rieder H., Schwab H.-J., Schwab K., Sinz E.J., Wittke M: Festschrift zum 60. Geburtstag von Walter Augsburger

Nr. 33 (1995) Sinz, E.J.: Kann das Geschäftsprozeßmodell der Unternehmung das unternehmensweite Datenschema ablösen?

Nr. 34 (1995) Sinz E.J.: Ansätze zur fachlichen Modellierung betrieblicher Informationssysteme - Entwicklung, aktueller Stand und Trends -

Nr. 35 (1995) Sinz E.J.: Serviceorientierung der Hochschulverwaltung und ihre Unterstützung durch workflow-orientierte Anwendungssysteme

Nr. 36 (1996) Ferstl O.K., Sinz, E.J., Amberg M.: Stichwörter zum Fachgebiet Wirtschaftsinformatik. Erscheint in: Broy M., Spaniol O. (Hrsg.): Lexikon Informatik und Kommunikationstechnik, 2. Auflage, VDI-Verlag, Düsseldorf 1996

Nr. 37 (1996) Ferstl O.K., Sinz E.J.: Flexible Organizations Through Object-oriented and Transaction-oriented Information Systems, July 1996

Nr. 38 (1996) Ferstl O.K., Schäfer R.: Eine Lernumgebung für die betriebliche Aus- und Weiterbildung on demand, Juli 1996

Nr. 39 (1996) Hazebrouck J.-P.: Einsatzpotentiale von Fuzzy-Logic im Strategischen Management dargestellt an Fuzzy-System-Konzepten für Portfolio-Ansätze

Nr. 40 (1997) Sinz E.J.: Architektur betrieblicher Informationssysteme. In: Rechenberg P., Pomberger G. (Hrsg.): Handbuch der Informatik, Hanser-Verlag, München 1997

Nr. 41 (1997) Sinz E.J.: Analyse und Gestaltung universitärer Geschäftsprozesse und Anwendungssysteme. Angenommen für: Informatik '97. Informatik als Innovationsmotor. 27. Jahrestagung der Gesellschaft für Informatik, Aachen 24.-26.9.1997

Nr. 42 (1997) Ferstl O.K., Sinz E.J., Hammel C., Schlitt M., Wolf S.: Application Objects – fachliche Bausteine für die Entwicklung komponentenbasierter Anwendungssysteme. Angenommen für: HMD – Theorie und Praxis der Wirtschaftsinformatik. Schwerpunkheft ComponentWare, 1997

Nr. 43 (1997): Ferstl O.K., Sinz E.J.: Modeling of Business Systems Using the Semantic Object Model (SOM) – A Methodological Framework - . Accepted for: P. Bernus, K. Mertins, and G. Schmidt (ed.): Handbook on Architectures of Information Systems. International Handbook on Information Systems, edited by Bernus P., Blazewicz J., Schmidt G., and Shaw M., Volume I, Springer 1997

Ferstl O.K., Sinz E.J.: Modeling of Business Systems Using (SOM), 2nd Edition. Appears in: P. Bernus, K. Mertins, and G. Schmidt (ed.): Handbook on Architectures of Information Systems. International Handbook on Information Systems, edited by Bernus P., Blazewicz J., Schmidt G., and Shaw M., Volume I, Springer 1998

Nr. 44 (1997) Ferstl O.K., Schmitz K.: Zur Nutzung von Hypertextkonzepten in Lernumgebungen. In: Conradi H., Kreutz R., Spitzer K. (Hrsg.): CBT in der Medizin – Methoden, Techniken, Anwendungen -. Proceedings zum Workshop in Aachen 6. – 7. Juni 1997. 1. Auflage Aachen: Verlag der Augustinus Buchhandlung

Nr. 45 (1998) Ferstl O.K.: Datenkommunikation. In. Schulte Ch. (Hrsg.): Lexikon der Logistik, Oldenbourg-Verlag, München 1998

Nr. 46 (1998) Sinz E.J.: Prozeßgestaltung und Prozeßunterstützung im Prüfungswesen. Erschienen in: Proceedings Workshop „Informationssysteme für das Hochschulmanagement". Aachen, September 1997

Nr. 47 (1998) Sinz, E.J.:, Wismans B.: Das „Elektronische Prüfungsamt". Erscheint in: Wirtschaftswissenschaftliches Studium WiSt, 1998

Nr. 48 (1998) Haase, O., Henrich, A.: A Hybrid Respresentation of Vague Collections for Distributed Object Management Systems. Erscheint in: IEEE Transactions on Knowledge and Data Engineering

Nr. 49 (1998) Henrich, A.: Applying Document Retrieval Techniques in Software Engineering Environments. In: Proc. International Conference on Database and Expert Systems

Applications. (DEXA 98), Vienna, Austria, Aug. 98, pp. 240-249, Springer, Lecture Notes in Computer Sciences, No. 1460

Nr. 50 (1999) Henrich, A., Jamin, S.: On the Optimization of Queries containing Regular Path Expressions. Erscheint in: Proceedings of the Fourth Workshop on Next Generation Information Technologies and Systems (NGITS'99), Zikhron-Yaakov, Israel, July, 1999 (Springer, Lecture Notes)

Nr. 51 (1999) Haase O., Henrich, A.: A Closed Approach to Vague Collections in Partly Inaccessible Distributed Databases. Erscheint in: Proceedings of the Third East-European Conference on Advances in Databases and Information Systems – ADBIS'99, Maribor, Slovenia, September 1999 (Springer, Lecture Notes in Computer Science)

Nr. 52 (1999) Sinz E.J., Böhnlein M., Ulbrich-vom Ende A.: Konzeption eines Data Warehouse-Systems für Hochschulen. Angenommen für: Workshop „Unternehmen Hochschule" im Rahmen der 29. Jahrestagung der Gesellschaft für Informatik, Paderborn, 6. Oktober 1999

Nr. 53 (1999) Sinz E.J.: Konstruktion von Informationssystemen. Der Beitrag wurde in geringfügig modifizierter Fassung angenommen für: Rechenberg P., Pomberger G. (Hrsg.): Informatik-Handbuch. 2., aktualisierte und erweiterte Auflage, Hanser, München 1999

Nr. 54 (1999) Herda N., Janson A., Reif M., Schindler T., Augsburger W.: Entwicklung des Intranets SPICE: Erfahrungsbericht einer Praxiskooperation.

Nr. 55 (2000) Böhnlein M., Ulbrich-vom Ende A.: Grundlagen des Data Warehousing. Modellierung und Architektur

Nr. 56 (2000) Freitag B, Sinz E.J., Wismans B.: Die informationstechnische Infrastruktur der Virtuellen Hochschule Bayern (vhb). Angenommen für Workshop "Unternehmen Hochschule 2000" im Rahmen der Jahrestagung der Gesellschaft f. Informatik, Berlin 19. - 22. September 2000

Nr. 57 (2000) Böhnlein M., Ulbrich-vom Ende A.: Developing Data Warehouse Structures from Business Process Models.

Nr. 58 (2000) Knobloch B.: Der Data-Mining-Ansatz zur Analyse betriebswirtschaftlicher Daten.

Nr. 59 (2001) Sinz E.J., Böhnlein M., Plaha M., Ulbrich-vom Ende A.: Architekturkonzept eines verteilten Data-Warehouse-Systems für das Hochschulwesen. Angenommen für: WI-IF 2001, Augsburg, 19.-21. September 2001

Nr. 60 (2001) Sinz E.J., Wismans B.: Anforderungen an die IV-Infrastruktur von Hochschulen. Angenommen für: Workshop „Unternehmen Hochschule 2001" im Rahmen der Jahrestagung der Gesellschaft für Informatik, Wien 25. – 28. September 2001

Änderung des Titels der Schriftenreihe *Bamberger Beiträge zur Wirtschaftsinformatik* in *Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik* ab Nr. 61

Bamberger Beiträge zur Wirtschaftsinformatik und Angewandten Informatik

Nr. 61 (2002) Goré R., Mendler M., de Paiva V. (Hrsg.): Proceedings of the International Workshop on Intuitionistic Modal Logic and Applications (IMLA 2002), Copenhagen, July 2002.

Nr. 62 (2002) Sinz E.J., Plaha M., Ulbrich-vom Ende A.: Datenschutz und Datensicherheit in einem landesweiten Data-Warehouse-System für das Hochschulwesen. Erscheint in: Beiträge zur Hochschulforschung, Heft 4-2002, Bayerisches Staatsinstitut für Hochschulforschung und Hochschulplanung, München 2002

Nr. 63 (2005) Aguado, J., Mendler, M.: Constructive Semantics for Instantaneous Reactions

Nr. 64 (2005) Ferstl, O.K.: Lebenslanges Lernen und virtuelle Lehre: globale und lokale Verbesserungspotenziale. Erschienen in: Kerres, Michael; Keil-Slawik, Reinhard (Hrsg.); Hochschulen im digitalen Zeitalter: Innovationspotenziale und Strukturwandel, S. 247 – 263; Reihe education quality forum, herausgegeben durch das Centrum für eCompetence in Hochschulen NRW, Band 2, Münster/New York/München/Berlin: Waxmann 2005

Nr. 65 (2006) Schönberger, Andreas: Modelling and Validating Business Collaborations: A Case Study on RosettaNet

Nr. 66 (2006) Markus Dorsch, Martin Grote, Knut Hildebrandt, Maximilian Röglinger, Matthias Sehr, Christian Wilms, Karsten Loesing, and Guido Wirtz: Concealing Presence Information in Instant Messaging Systems, April 2006

Nr. 67 (2006) Marco Fischer, Andreas Grünert, Sebastian Hudert, Stefan König, Kira Lenskaya, Gregor Scheithauer, Sven Kaffille, and Guido Wirtz: Decentralized Reputation Management for Cooperating Software Agents in Open Multi-Agent Systems, April 2006