Secure, Trusted and Privacy-aware Interactions in Large-Scale Multiagent Systems

Federico Bergenti
Dipartimento di Ingegneria dell'Informazione
Università degli Studi di Parma
Parco Area delle Scienze 181/A, 43100 Parma, Italy
Email: bergenti@ce.unipr.it

***Abstract*— One of the inherent problems of large-scale, open multiagent systems is the lack of mechanisms and tools to guarantee legally valid interactions. Agents are supposed to perform crucial tasks autonomously and on behalf of humans; however, *(i)* they are not legal persons on their own, and *(ii)* of a full legal corpus for the virtual world and its inhabitants is yet to come. Therefore, the ultimate responsible for the actions of an agent is its developer. In this paper we address an innovative model of interaction between agents that leads to an increase of the level of security and trust in privacy-aware, interaction-intensive multiagent systems. In particular, after a brief introduction, we focus in Section II on some common problems related to trust and security in real-world, liable interactions. In Section III, we address these problems and outline some abstractions that we use to guarantee a sound level of security and privacy-awareness in interactions with third-party (possibly unknown) agents, whether human or not. Then, in Section IV we describe the design of an API that we implemented to provide developers with a general-purpose, reusable means to realize secure, trusted and privacy-aware multiagent systems. To conclude, in Section V we briefly discuss our model and outline directions of future development.**

I. Introduction

Agent technology is quickly evolving towards the realization of complex societies of agents. Just to cite one recent example, the aims and scope of the IST project CASCOM [4] show how agents are becoming more and more relevant in important sectors, e.g., healthcare and personal data management. This evolution is not yet matched by an equivalent legal development. The lack of a legal substrate capable of grounding the interactions between agents ultimately means that every aspect of interactions (e.g., see [11]) with other (possibly unknown) third-party agents must be explicitly treated by the developer. Moreover, for a legal point of view, the developer is the ultimate responsible for the actions of its agents. This situation is then exacerbated by the impossibility of tracing all actions agents perform: if we cannot guarantee traceability [19] of the actions of individual agents, no law would be sufficient to prevent and punish mendacious agents (whether human or not). Obviously, traceability does not guarantee that agents could not misbehave; anyway, if they do so, other agents would have the possibility of demonstrating the misbehaviour.

Having said this, the ultimate goal of our work is to provide mechanisms and tools to support agents in interacting:

1) In a secure, traceable and privacy-aware way; and
2) With guarantees of a desired level of security and trust, exploiting the minimum possible number of trusted parties.

Our study of these issues is concretized in the realization of a model capable of representing a secure, trusted and privacy-aware interaction between two agents. The generalization of this model to multi-party interactions is quite straightforward, but its exhaustive description is out of the scope of this paper.

Our work is based on the introduction of two closely-related abstractions, *Validation-Oriented Ontologies* (*VOOs*) and *Guarantors*. A VOO is a signed set containing an ontology [1] and all runtime tools needed to assert that a particular individual of the world actually belongs to a certain family of individuals. Guarantors are agents that, in some sense, play the role of middleman in interactions. Guarantors are trusted by all interacting parties and they are in charge of supporting interactions by providing (under their responsibility) all necessary VOOs.

This paper is organized as follows: next section describes some crosscutting problems of two-party interactions and it motivates why we need something more that available techniques and tools to guarantee security, trust and privacy in multiagent systems. Section III briefly describes our model and introduces the notions of VOOs and Guarantors. Section IV shows how we support our model and its new abstractions by describing an API that we realized to support developers in their everyday work. Finally, Section V briefly discusses our model to point some interesting direction of development and to show its wide applicability in real-world scenarios.

II. Problems in Two-Party Interactions

The first assumption that we take in the discussion of our model is that, from the point of view of security, trust and privacy, we can always reduce any two-party interaction to the act of signing of a contract. Then, we assume that proposals and agreements between interacting parties are exchanged in the form of individuals of known ontologies. This assumption allows agents to manage the information contained in proposals and agreements in a friendly way, e.g., to reason about proposals and to assert the formal validity of proposals against the constraints of the ontology.

All in all, the assumption of modelling interactions as contracts that are individuals of known ontologies is ab-

solutely general and has some remarkable advantages. The most interesting advantage that we see is the possibility of combining simple ontologies into complex models of proposals and agreements. We can compose simple ontologies into complex descriptions of proposals and agreements, thus avoiding duplication of definitions and possible ambiguities.

The second advantage that we see in our working assumption is that it greatly simplifies the creation and validation of proposals and agreements. The creation of a proposal is reduced to the creation of one or more individuals of known ontologies, with properties set accordingly to given values (potentially specified in external policies). Controlling the suitability of a proposal simply reduces to checking whether a candidate proposal actually belongs to the family of admissible proposals described in the referenced ontology.

Finally, ontologies expressed in common formats are easily mapped into human readable documents for a subsequent inspection of the agreements that software agents may have autonomously signed.

A. Problem 1. Trusting Ontologies

Ontologies seem to be a suitable means for describing agreements, but any attempt to use them in real-world scenarios immediately encounters a problem: How an agent could trust a new ontology? Suppose that a seller of bandwidth requires to negotiate agreements with potential customers using an ontology available in some public repository. This ontology may model some property as being "required by local laws." How could customers trust this requirement if they have no trust relationship with the seller that pointed it to this ontology? Could a customer (in some sense) validate the ontology to decide whether to trust it or not?

Another facet of this problem occurs in the case of an ontology that is partially non-disclosed. Let us suppose that the aforementioned seller creates its ontology and splits it into two parts: a public part describing valid proposals and agreements, and a private part used to model the policies that it employs to enforce bandwidth reservations, i.e., the policies that it uses to reason on proposals. This last part contains background knowledge on the marketing strategies of the seller and it is vital not to disclose this knowledge to potential competitors. In this case, a full fledged reasoning on the ontology could be done only by accessing the whole ontology, and only partial reasoning is possible for customers.

Moreover, we have to take into account a third (very serious) facet of this problem: there is no way to validate the adherence of the ontology to real-world laws, without involving highly specialized jurists. Obviously, no potential customer would be in the position of performing this sort of validation.

In the end, all these exemplified facts of the same problem, i.e., trusting the ontology, show that trust cannot be given to an ontology per se: it must be accorded to its signer. Ontologies used to model formal agreements and contracts must be provided by trusted and liable signers.

B. Problem 2. Trusting Identities

The ultimate aim of our model of interaction is to guarantee legal validity. Therefore, the problem of checking the identities of involved agents is obviously critical. Unfortunately, a simple static control of identities by means of certificates [6], [7] is inadequate because, e.g., certificates can be revoked or keys can be stolen. This inadequacy should not be surprising because it is very common also in human interactions. The identification of agents in a secure, trusted and privacy-aware multiagent system can be performed only through a set of runtime tools capable of validating certificates, and thus realizing a trusted source of identification.

The problem of checking identities is closely related to the representation of identities. The identification code is the only means that we have to validate the identity of a legal person (physical or not). Therefore, one of the very basic issues that we have to tackle is how to represent identities in an agent-processable way. In our model, we decided to design an ontology describing legal persons and their attributes and to associate this ontology with a set of general-purpose tools for addressing the majority of problems related to identification. The connection between this ontology and its tools is reinforced by the necessity of a common trusted signer.

It is worth noting that in order to fully exploit the possibility of having runtime tools capable of providing some sort of guarantees regarding sensible tasks on an ontology, both the ontology and its associated tools must have the same levels of trust and security. Let us consider these two examples to clarify this point.

1) Case 1. Trusted ontology - Untrusted tools: Suppose that two negotiating agents trust the same ontology (i.e., they trust the publisher of the ontology) but they use untrusted services to perform validations. They exchange proposals until an agreement is reached and they mutually check their identities using an untrusted tool. Since they do not trust the identity verification tool, they can both suppose that they are signing an agreement with an unknown party.

2) Case 2. Untrusted ontology - Trusted tool: Suppose that we have an identity-verification tool that receives in input an ontology and an identity, and verifies the identity in a database. What happens if someone gives formally valid (compliant with the ontology), but legally void identity? Since the given identity matches the record in the database of identities, the tool would return an affirmative answer, but this identity is legally void and therefore unusable in signing formal agreements.

These two examples show that both ontology and runtime tools must be trusted and secure. If any of the two has not a suitable level of trust and security, the combined use of them will not result in a secure and trusted interaction.

III. Trust, Contracts and Guarantors

The analysis of the two-party interaction outlined in the previous section allows to introduce two abstractions that

we can generally use to model secure, trusted and privacy-aware interaction between two agents. These closely-related abstractions, namely Validation-Oriented Ontologies (VOOs) and Guarantors, are briefly described in this section.

The problem of defining trust has been addressed in many different ways [15]. While we recognize the importance of cognitive models [5] to quantify trust, we start from the definition given in [9] to provide a probabilistic interpretation of trust. In particular, if *"Trust is the subjective probability by which an individual, A, expects that another individual, B, performs a given action on which its welfare depends"*, it is reasonable to model trust in terms of an estimation of the real probability by which B would perform the target action. Many factors contribute to this estimation [11], [13]; nonetheless we use a blackbox approach, in which trust is modelled as a random variable in an interval $[p_{a,x}, p_{b,x}]$. The only assumption that we take is that such an estimation is a reasonable approximation of the real value of the quantity.

Our probabilistic model of trust is out of the scope of this paper, and we simply enumerate the quantities that we exploit in our treatment:

1) $p_{k,x}$, the probability that the information k provided by agent X is correct;
2) $p_{c,x}$, the probability that agent X would adhere to all the obligations stated in contract c;
3) $t_{c,x,y}$, the level of trust agent X has in agent Y with respect to contract c;
4) $p_{a,x}$, the minimum value of trust in an estimation, i.e., the lower bound of the probability distribution function of trust;
5) $p_{b,x}$, the maximum level of trust in an estimation, i.e., the upper bound of the probability distribution function of trust.

Since trust expresses the estimate of a probability, it is clear that p_a and p_b are both between zero and one. The assumption that $p_b \geq p_a$ is not restrictive.

As stated in the previous section, we assume that all interactions between two agents can be reduced, from the point of view of trust and privacy, to the action of signing a contract. While it is reasonable to think of a number of different (and very complex) contracts [2], we adopt a very simple contract model. It involves only two signers, and it is totally described by two triples, one known to each signer. Each triple, that we call *subjective evaluation* of the contract, is made of a *reward*, an *investment* and a *penalty*. This triple summarizes the contract, and its effects, for the agent to which the triple belongs. Being subjective values, it is not possible to assess any mathematical relation between values of two different subjective evaluations, even though they refer to the same contract.

The subjective evaluation of contract c given by agent X is written as follows:

1) $R_{c,x}$ is the reward that agent X receives upon success of contract c;
2) $I_{c,x}$ indicates the investment that agent X makes in contract c; i.e., a certain granted value that it renounces to, when signing contract c;
3) $P_{c,x}$ is the penalty of the contract, i.e., the value that agent X receives if the contract fails because of the other party.

The contract gives to its signers the absolute security of receiving the stated values, i.e.:

1) if the contract is respected, agent X receives $R_{c,x}$ with probability one;
2) if the contract fails because of agent Y, agent X receives $P_{c,x}$ with probability one.

Another assumption that we take concerns the order between reward, investment and penalty in a subjective evaluation. We are interested in contracts whose parameters are ordered as follows:

$$P_{c,x} \leq I_{c,x} \leq R_{c,x} \tag{1}$$

This inequality expresses the fact that contracts are advantageous but risky. This, in turn, implies that an agent signs a contract in the hope that it would be respected by the other signer, since in case of failure it would experience the following loss:

$$I_{c,x} - P_{c,x} \tag{2}$$

Furthermore, each agent does not consider its own failure probability, since it will only consider contracts that it can reasonably respect; nevertheless the uncertainty about the other signer remains.

Taking this probabilistic model that we briefly described here, and that is subject for an in-depth investigation in a future paper, we can provide a probabilistic description of a two-party interaction. From the point of view of security, trust and privacy, such an interaction can take advantage of the presence of a Guarantor that plays (in some sense) the role of middleman in the interaction. In order to provide a synthetic description of the aims and scope of the notion of Guarantor, we need to introduce another accessory abstraction, namely Validation-Oriented Ontology.

A *Validation-Oriented Ontology* (VOO) is a signed set containing:

1) An ontology that models a domain;
2) A set of runtime tools capable of asserting properties of individuals of this ontology.

Runtime tools are intended to provide a means for validating assertions on the domain described by the ontology without requiring a full-fledged reasoning on the domain. As we have seen in the previous section, this is essential from the point of view of security and trust for real-world applications.

One very important advantage of the introduction of VOOs is that they reduce the amount of distributed trust, since in a single signed object lay both the semantic description of thing and a set of related actions.

Moreover, VOOs promote software reuse and help standardization, since many ontology-related tasks are performed from external bodies (the tools of VOO) in a standard, well-defined, trusted and secure way. It is worth noting that the

concrete technology used to realize the tools of the VOO is not mandatory: they could be Web or Grid services [8], as well as RMI invocations, as long as they are projected and signed together with their ontology. In this way it is possible to achieve platform independence by including in the VOO a description of the invocation procedure of its tools.

VOOs are not sufficient to address all issues related to real-world agreements because we need to trust both the VOO and the signer of the VOO, as discussed in the previous section. In fact, if we go back to the human world, the proper way to stipulate contracts is through a notary public. This happens because only legal person trusted by the State can perform critical tasks (e.g., querying databases containing privacy-critical information). This is the reason why we introduce the abstraction of Guarantor, and we say that an agent is a Guarantor for an interaction between two other agents if it can sign a VOO that the two other agents can use in their interaction.

We can be more precise in this definition by rephrasing the auditing principle of [2] for a validation case:

If Role 1 cannot witness the truthfulness of an assertion about Role 2, another Role 3 should testify the condition of Role 2 if the party playing Role 2 is not trusted by the party playing Role 1. This document must be received by Role 1 before the execution of its primary activity, and the party playing Role 3 should be trusted by the party playing Role 1.

According to this principle, we suppose that both agents involved in a two-party interaction trust a common agent, playing Role 3, that we call Guarantor. This agent is supposed to be responsible for the exactness of the information provided by itself and by its tools. Unlike other agents, the Guarantor of the interaction can easily check ontologies, tools and other Guarantors, to provide tools that can operate on other Guarantors' certificates, ontologies, etc. Therefore, the introduction of the Guarantor allows agents to put their trust in a single entity, thus simplifying greatly the decisions related to according or revoking trust.

In summary, in our model the Guarantor is responsible for the following tasks:

1) Provide identity certificates;
2) Provide signed ontologies compliant with real-world laws;
3) Provide signed runtime tools for its ontologies and/or certifying external tools under its responsibility.

Then, if we remember that identity certificates are provided as signed instances of concepts of an ontology, and if we go back to the previous definition of VOO, these three responsibilities of the Guarantor can reduce to a single responsibility: *provide VOOs*.

The Guarantor takes the responsibility of catalyzing the trust of an interaction in various ways, e.g., through:

1) A signed list of trusted tools;
2) A certified public key whose private key is provided only to trusted tools;
3) A certified set of services that could access the Guarantor's database and whose use could be detected by the tools' user.

IV. An API for Secure, Trusted and Privacy-Aware Communications

The abstractions of VOOs and Guarantors must be adequately supported by some development tools in order to implement them correctly in real-world MASs. This is the reason why we developed an API for JADE capable of providing a direct support to developers in the realization of secure and trusted MASs.

The API we developed focuses primary on the double-Guarantor model, because it is general enough to subsume the single-Guarantor model, but it is simple enough to allow an in depth evaluation and study. Furthermore the double-Guarantor model is probably the most frequent case.

We designed our API to match a set of fundamental requirements, that resulted in strict development guidelines.

1) Privacy. All communications must be encrypted and directed to trusted parties.
2) Traceability and security. Invocations involving tools of VOOs must be signed by the caller, while responses from such tools must be signed, directly or indirectly, by a Guarantor. The API transparently checks this property and provides a transparent tracing service that logs all invocations and responses.
3) Locality. The number of trusted parties involved in any interaction must be kept at minimum. This means that an operation performed by a given Guarantor in a mutual recognition case, must be delegated upwards and not delegated immediately to the other Guarantor's tools.
4) Transparency. The invocation procedures of VOO tools must be transparent to the user, i.e., the user is not directly involved in the use of the tools that the VOO provides.
5) Ease of use. The API must provide high level procedures to perform common tasks, as well as low level, more specific procedures devoted to fine-grained (and less common) tasks.
6) Standardization. Information exchange, including certificates and proposals, must be performed using well-known formats.

These guidelines are completed with the following use cases and result in a first set of requirements that we used in the realization of our API.

The design of the API is split into two views: *(i)* a client view that shows the classes that a client agent can use to access the services of the security and privacy subsystem, and *(ii)* a Guarantor view that describes the components that the Guarantors use to implement their functionality. Such views are connected through the *Guarantor* interface, that plays the logical role of a remote interface that Guarantors implement and that client exploit by means of proxies.

```
public interface Guarantor {
    public SessionToken signOn(Credentials c) throws SignOnFailed;
    public boolean signOff(SessionToken st); /* true on successful sing-offs */

    public Object directInvocation(
        DistributedTimeStamp dts,
        ServiceDescriptor sd,
        Object[] parameters
    );

    public DelegationToken createDelegationToken(
        ServiceDescriptor sd,
        DelagationDescriptor dd
    );

    public Object indirectInvocation(
        DistributedTimeStamp dts,
        DelegationToken dt,
        Object[] parameters
    );
}

public interface ServiceDelagationDescriptor {
    public Certificate delegator();
    public Certificate delegated();

    public long getNumberOfInvocations(); /* max number of invocations */
    public long getDeadline(); /* in millis from generation time */
}

public interface Token {
    public String getCanonicalString(); /* UUencoded */
    public long getExpirationDate(); /* in millis from generation time */
}

public interface SessionToken extends Token {}

public interface DelegationToken extends Token {}
```

Fig. 1. Client view of the API

It is worth noting that the client view represents a mandatory interface, on the contrary, the Guarantor view is only a suggestion of a possible internal design of Guarantors. Obviously, client view plays a substantially more important role in this design.

A. Client View

For the sake of clarity and readability, Figure 1 collects the interfaces of the client view in terms of Java interfaces.

The central interface of the client view is Guarantor. It encapsulates all methods that Guarantors expose to clients. Such methods are accessed remotely through an encrypted channel and they are available after an initial mutual recognition phase through the *signOn()* method. Client wishing to use the services of a Guarantor, invoke this method and pass their credentials. The type of requested credentials depends on the Guarantor: simple username/password may be sufficient in certain cases, or more complex X.509 certificates may be needed in other cases. Guarantors will provide their specific subclass of interface Credentials to have clients provide the required information.

If the Guarantor intends to serve the client that issued the *signOn()* request, it will respond with a *SessionToken* that the client will use for subsequent invocations on the Guarantors interface. A *SessionToken* is a particular sort of *Token*. Just like all tokens, it has an expiration date and it can be converted in a canonical string.

signOn() requests are invoked on some kind of secure channel, e.g., HTTPS, and subsequent services are requested on the same secure channel. A client can issue request for services until the client itself signs off (through the *signOff()* method) or until the session expires.

Once a client is authenticated with a Guarantor, it can perform two kinds of requests:

1) Direct requests, i.e., requests for services whose outcome is used by the client itself;
2) Indirect requests, i.e., requests that are performed on behalf of some other client.

Direct requests are performed simply through the *directInvocation()* method. These are ordinary requests for services except for the following two constraints:

1) Parameters and result value are transported on a secure channel;
2) The Guarantor is responsible for tracing the request to guarantee non-repudiability;
3) The client is responsible for providing a distributed timestamp to allow for traceability of complex interactions.

The *directInvocation()* method is the only mechanism that Guarantors offer to have services performed in this way. Other methods of the Guarantor interface (or of any of its subclasses) are not guaranteed to respect the aforementioned constraints. All in all, the *directInvocation()* method plays the role of the Dynamic Invocation Interface of CORBA objects, or of the *Method.invoke()* method of Java reflection. It is the preferred way to handle secure services.Indirect requests are a delegation mechanisms that allows a client (A, delegated) to have a service performed on behalf of another client (B, delegator). This process is made of the following steps:

1) Client A requests the Guarantor to grant indirect requests to client B;
2) If the Guarantor can honour this request, it will accept requests from B and serve them as if they were requested by A;
3) The Guarantor stops serving indirect requests from B if the delegation has expired, e.g., because the maximum number of requests from B is reached.

The first step of this process is performed when A invokes the *createDelegationToken()* method on the Guarantor interface. This method needs the following parameters:

1) A ServiceDescriptor that identifies which service(s) of the Guarantor the client is willing to delegate;
2) A DelegationDescriptor that provides the Guarantor with all information needed to actually perform the delegation, e.g., who is the delegated client, for how long the delegation will last.

If the Guarantor can grant the delegation of the service to B, the return value of *createDelegationToken()* is a globally unique token that identifies the delegation. The delegated client B will use this token to finally access the services of the Guarantor through a call to *indirectInvocation()*. This method has exactly the same meaning and constraints of the *directInvocation()* method, except for the fact that it can be invoked by delegated clients that are not currently authenticated with the Guarantor.

If a Guarantor needs additional, application-specific information, to grant indirect requests, it can provide its own sub-

```
public interface SensitiveDataStore {
    public SessionToken signOn(Credential c) throws SignOnFailedException;
    public boolean signOff(SessionToken st); /* true on successful sing-offs */

    public ResultSet query(SessionToken st, QueryStatement q) /* result set */
        throws IllegalSessionToken, IllegalStatement;
    public long insert(SessionToken st, InsertStatement o) /* number of additions */
        throws IllegalSessionToken, IllegalStatement;
    public long update(SessionToken st, UpdateStatement o) /* number of updates */
        throws IllegalSessionToken, IllegalStatement;
    public long delete(SessionToken st, DeleteStatement o) /* number of deletions */
        throws IllegalSessionToken, IllegalStatement;
    public boolean create(SessionToken st, CreateStatement o) /* creations */
        throws IllegalSessionToken, IllegalStatement;

    public void setUsernameForNotifications(SessionToken st, String username)
        throws IllegalSessionToken, IllegalUsername;
}

public interface InvocationTracer {
    public void traceIndirectInvocation(
        DistributedTimeStamp dts,
        DelegationToken dt,
        Object[] parameters,
        Object result
    );

    public void traceDirectInvocation(
        DistributedTimeStamp dts,
        ServiceDescriptor sd,
        Object[] parameters,
        Object result
    );
}
```

Fig. 2. Guarantor view of the API

classes of classes *ServiceDescriptor* and *DelegationDescriptor* interfaces.

Indirect requests is the preferred way to allow a third party having a service done without explicitly requesting sensitive data. For example, let's consider a buyer A and a seller B. Normally, the seller will request the details of As credit card in order to:

1) Check the validity of the credit card;
2) Perform the withdrawal of the exact amount of the requested payment.

B would be able to perform exactly such operations if buyer A would instruct its bank (the Guarantor) to serve this two requests from B as if they were issued by A itself. This approach has the great advantage of allowing A to buy from B without revealing any sensitive information. The delegation token that allows B to perform the withdrawal is a sort of anonymized view of the sensitive data of A. Formally, this delegation token is a *one-time password* for logical access control.

B. Guarantor View

The Guarantor view of the architecture describes how a Guarantor may implement a general-purpose infrastructure for providing its services with requested level of security and privacy. This architecture is not mandatory because every Guarantor may decide its own optimized approach to provide services. Anyway, the quality of Guarantors in performing tasks related to security and privacy, e.g., the global uniqueness of the generated tokens, or the correct tracing of invocations, are important metrics for clients to put trust of Guarantors. Therefore, the Guarantor view is highly recommended as it helps clients estimating the reputation of Guarantors. Figure 2 shows the Java interfaces that make the Guarantor view of the architecture.

One of the principal interfaces that build the Guarantor view of the architecture is *SensitiveDataStore*. This is an abstract view of a data store that is meant to allow for a seamless treatment of sensitive data. It is worth remembering that every Nation in the European Community adopted laws to provide guarantees to citizens regarding the treatment of their sensitive data.

Such laws are all rooted in a note of the European Commission and they all contain strict technical requirements that databases of sensitive data must follow. As an example, the following are examples of the requirements of the Italian law on privacy:

1) The password of the manager of the data store must be of 8 alphanumerical characters, at least;
2) The password of the manager of the data store must be changed every 3 months;
3) If any access credential to the data store has not been used for more than 3 months, it must be revoked.

Any implementation of the *SensitiveDataStore* interface will wrap existing technologies for storing data, e.g., JDBC or JNDI, and it will add the support for any requirement to make it compliant with a particular legislation (at a particular time). Any *SensitiveDataStore* allows a direct management of the data it contains thought methods: query, insert, update, delete and create. These are wrapper to the underlying storing technology and their statement parameters are concrete subclasses that provide all necessary information to concretely perform requested operations. Not all such methods are always permitted to allow for accommodating different levels of management of the data store, e.g., query may be always possible, but creation and deletion of database table is possible only when the Guarantor is not online. Any attempt of violating such application-specific constraints will generate an *InvalidStatement* exception.

Methods *setUserNameForNotifications()* is used to instruct the data store to actively provide information on compliancy. For example, a particular data store may decide to notify the manager of any credential that no longer complies with laws, or it may decide to notify the administrator of any clean-up of old data. This unusual behaviour of a *SensitiveDataStore* is needed to allow data stores suspending operations on a session when, for some reason, the session does no longer comply with laws. So, for example, a properly implemented sensitive data store would stop functioning when the password of the person in charge is more than 3 months old.

The second interface that may be used to design Guarantors is *InvocationTracer*. This interface provides all methods for tracing direct and indirect requests served or rejected by the Guarantor. Such requests are stored in a *SensitiveDataStore* that would save all (context) information regarding requests, It is worth noting that the invocation *DistributedTimeStamp* property allow to correlate logs of different Guarantors, and therefore it allows backword tracing the execution of complex actions.

V. Discussion

The central focus of this paper is on the motivated introduction of two abstractions, VOOs and Guarantors, that we can use to provide general-purpose mechanisms to realize secure and trusted MASs. The need of these abstractions should be clear if we go back to the very general issues related to security and trust that we identified for two-party interactions. Obviously, the introduction of these abstractions is not the only way we can think to tackle such issues, but we believe that our approach has two interesting properties:

1) Concentrated trust. Guarantors are sorts of trust catalysts that we use to keep trust concentrated on the minimum number of parties. From the point of view of interacting agents, this is good because the number of operations related to according or revoking trust is minimized.
2) Pragmatic interactions. The strict coupling between an ontology and a set of tools capable of performing general-purpose, critical tasks on the individuals of this ontology (i.e., the idea of VOO) guarantees the possibility of performing secure and trusted interactions also to agents with minimal reasoning capabilities.

In conclusion, we believe that the introduction of VOOs and Guarantors provides a solid ground for the concrete development of trusted and secure MASs. Many issues related to these properties are encapsulated by these abstractions and we believe that their in-depth study can lead to a better understanding of the subtle behaviours of these complex systems in real-world situations.

Acknowledgements

This work is partially supported by project CASCOM (FP6-2003-IST-2/511632). The CASCOM consortium is formed by DFKI (Germany), TeliaSonera AB (Sweden), EPFL (Switzerland), ADETTI (Portugal), URJC (Spain), EMA (Finland), UMIT (Austria), and FRAMeTech (Italy). This article reports on joint work that is being realised by the consortium. The authors would like to thank all partners for their contributions.

References

[1] Baader, F., Calvanese, D., McGuinness, D., Nardi, D., and Patel-Schneider, P.F., (Eds.), *The Description Logic Handbook: Theory, Implementation and Applications*, Cambridge University Press, 2003.
[2] Bons, R.W.H., *Designing Trustworthy Trade Procedures for Open Electronic Commerce*, Ph.D. Dissertation, 1997, EURIDIS and Faculty of Business Administration, Erasmus University.
[3] Casati, F., Shan, E., Dayal, U., and Shan, M.-C., *Service-Oriented Computing: Business-Oriented Management of Web Services*. Communications of the ACM, 46:10, October 2003.
[4] CASCOM Web site `http://www.ist-cascom.org`
[5] Castelfranchi, C., and Falcone, R. Principles of Trust for MAS: Cognitive Anatomy, Social Importance, and Quantification. In Proceedings of *The International Conference of Multi-agent Systems (ICMAS)*, 72–79, 1998.
[6] Ellison, C., *SPKI Requirements*. IETF RFC 2692, September 1999.
[7] Ellison, C., Frantz, B., Lampson, B., Rivest, R., Thomas, B., and Ylonen, T., *SPKI Certificate Theory*. IETF RFC 2693, September 1999.
[8] Foster, I., Kesselman, C., and Tuecke, S., *The Anatomy of the Grid: Enabling Scalable Virtual Organizations*, Int'l Journal of Supercomputer Applications, 15(3), 2001.
[9] Gambetta, D. (Ed.), *Trust: Making and Breaking Co-operative Relations*, Basil Blackwell, Inc., UK, 1985.
[10] JENA Web site `http://jena.sourceforge.net`
[11] Jennings, N. R., Parsons, S., Sierra, C. and Faratin, P., *Automated Negotiation*, in Procs. of the 5^{th} Int'l Conference on the Practical Application of Intelligent Agents and Multi-Agents Systems, PAAM-2000, Manchester, UK.
[12] Martin, D., Paolucci, M., McIlraith, S., Burstein, M., McDermott, D., McGuinness, D., Parsia, D., Payne, T., Sabou, M., Solanki, M., Srinivasan, N., and Sycara, K., *Bringing Semantics to Web Services: The OWL-S Approach*, in Procs. of the 1^{st} Int'l Workshop on Semantic Web Services and Web Process Composition (SWSWPC 2004), July 2004, San Diego, USA.
[13] Marsh, S. *Formalising Trust as a Computational Concept*. Ph.D. diss., Department of Mathematics and Computer Science, University of Stirling, Stirling, UK, 1994.
[14] Meyer, B., *Object Oriented Software Construction, Second Edition*, Prentice-Hall, NJ, 1997.
[15] MINDSWAP, *A Definition of Trust for Computing with Social Networks* Technical report, University of Maryland, College Park, February 2005.
[16] OWL Web site `http://www.w3.org/2004/OWL`
[17] Poggi, A., Tomaiuolo, M., Vitaglione, G., *Do Agents Need Certificates? Distributed Authorization to Improve JADE Security*, in Procs. of the 6^{th} Int'l Workshop on Trust, Privacy, Deception, and Fraud in Agent Societies, AAMAS 2003, July 2003, Melbourne, Australia.
[18] Racer Web site `http://www.sts.tu-harburg.de/~r.f.moeller/racer/`
[19] Szomszor, M., and Moreau, L., *Recording and reasoning over data provenance in web and grid services*, in Procs. of the Int'l Conference on Ontologies, Databases and Applications of Semantics (ODBASE'03), LNCS 2888, November 2003, Catania, Italy.