Secure Knowledge Query Manipulation Language: a Security Infrastructure for Agent Communication Languages

Muhammad Rabi Tim Finin Alan Sherman
Yannis Labrou
Computer Science and Electrical Engineering
University of Maryland Baltimore County
Baltimore, Maryland 21228-5398

Abstract

Despite the security and privacy concerns that agents could encounter whenever they cross multiple administrative domains, the agent communication languages standards lack the necessary constructs that enables the secure cooperation among software agents. We propose Secure Knowledge Query Manipulation Language (SKQML) as a security infrastructure for KQML-speaking agents. SKQML enables KQML-speaking agents to authenticate one another, implement specific security policies based on authorization schemes, and whenever needed ensure the privacy of the messages exchanged.

SKQML is simple, extensible, and at a level appropriate for intelligent communicating agents. In addition, SKQML employs public-key cryptographic standards and it provides security mechanisms as an integral part of the communication language. This paper details the synthesis of public key certificate standards and agent communication languages to achieve an infrastructure that meets the security needs of cooperating agents with detailed examples using a partial prototype implementation of this infrastructure.

1 Introduction

With the proliferation of the Internet and the World-Wide-Web, software agents are set to become the foundation for Web-based services. Intelligent agents are being built for a wide range of problem domains including document and information retrieval, hight performance scientific computing, distributed network management, and electronic commerce just to name a few.

Although distributed agent-based systems that support collaborative problem solving encounter security and privacy concerns especially when they cross multiple administrative domains, one of the most important infra-structural issues, security, has not been fully addressed in the context of agent environment. In the following sections, we provide a security infrastructure for agent communication languages. For two agents to communicate with each other by exchanging messages, they must agree on the syntax and semantic of these messages. Agent communication languages (ACL) for instance KQML [1, 2, 3, 4] and FIPA ACL [5] are languages with precisely defined syntax, semantics and pragmatics that is the basis for communication among autonomous software agents. Despite the availability of many security approaches, products, and tools, a consistent widely adopted, and cost-effective solution must be found for a security infrastructure in agent environments. Security mechanisms must be included as an integral part of these environments. Attaching security mechanisms to already built agent environments as "add-ons" will introduce more problems of interoperability, integration, and usability.

We employ public-key cryptographic objects in defining an infrastructure for agent communication languages. We begin by identifying the security functional requirements for agent communication languages including authentication, authorization, and privacy. Furthermore, security functions must be offered at the communication language message level even though it could be achieved through lower level layers such as transport or network layers. This would ensure that agents will focus on implementing their own security policies instead of dealing with low-level details interacting with lower layers. We show that the proposed architecture satisfy those requirements by providing means to define groups, issue group membership certificates, enable authentication of agents, provide authorization based on access control lists, and provide means to ensure message privacy.

In the following sections, we propose Secure Knowledge Query Manipulation Language (SKQML) as an extended KQML. First, we introduce three new performatives that facilitate the implementation of agent security policies. SKQML security performatives are based on existing proposals for public-key infrastructure which includes: IETF Simple Public Key Infrastructure (SPKI), Distributed Trust Management [6], and Rivest and Lampson [7] proposal on Simple Distributed Security Infrastructure (SDSI/SPKI) and it is based on earlier work by Thirunvukkarasu, Finin, and Mayfield [8]. Second, we define a propositional security language that is based on public-key certificate standards thus interoperability and integration with other trust management engines can be easily achieved. Third, we introduce new protocols for trust management with examples from a prototype demo system that is based on a university setting environment.

Finally, we end with conclusions and suggestions for further research.

2 Security functional requirements

The decentralized peer-to-peer nature of agent-based application requires a solution to the trust management problem identified in [6]. In this section, we summarize the functional requirements as well as capabilities as proposed in [8] and identify new ones, they include:

- Authentication of principals. Agents should be capable of proving their identities to other agents as well as verify the identity of other agents.
- Security of communication between agents which may require authentication of agent identities. It also requires message integrity and optionally confidentiality and protection of messages in transit.
- Preservation of message integrity. Agents should be able to detect intentional or accidental corruption of messages.
- Detection of message duplication or replay. A rogue agent may record a legitimate conversation and later play it back to disguise its identity. Agents should be able to detect and take corrective measures to prevent such playback security attacks.
- Non-repudiation of messages. An agent should be accountable for the messages that they have sent or received, i.e., they should not be able to deny having sent or received messages.
- Prevention of message hijacking. A rogue agent should not be able to extract the authentication information from an authenticated message and use it to masquerade as a legitimate agent.
- Security auditing that will allow agents to be identified correctly under all circumstances.

The security infrastructure for KQML must also satisfy the following requirements: Independence of KQML performative and the application semantic, simplicity, independence of transport layer, independence of global clock or clock synchronization, authentication by crypto-unaware agents, and finally, support for a wide variety of cryptographic systems and standards. The security infrastructure of the KQML must support delegation. An agent must be able to delegate one or more of its capabilities to one or more agents. With delegation comes the need for agents to define groups of agents as well as the ability to define access control within these groups.

3 Agent Security Architecture

Before introducing SKQML, we must define we mean by “agent-identity” and “agent-name” as well as the binding that exists between them.

3.1 Naming Agents

Agents identities are tightly bound to agents names. Finin, Potluri, and others in [9] recognized the need for agents to be named and provided a solution based on *Agent Domains*. One obvious requirement for naming agents is that agents must have names that are independent of any implementation details (i.e. transport mechanisms, IP address, port numbers, etc.). In their solution to the agent-naming problem, Finin et al. proposed the use of agent domains, which are organized into an agent domain hierarchy. Agent names resolution will be performed by *agent-name-server* agents that use a distributed protocol similar to that used by the Internet domain name servers (DNS).

Their proposal doesn't require the addition of any new KQML performatives or parameters. To authenticate agents, their agent naming proposal must be extended. One approach to achieve that is to add constructs to the Agent Name Server protocol in a fashion similar to that proposed in the new DNSSEC protocol [10].

Recently, members of the Jackal project provided another solution to the agent-naming problem [11]. Jackal is a Java implementation of the KQML agent communication language environment. Jackal uses a hierarchical naming scheme for names that are unique across time and space and it utilizes Fully Qualified Agent Name (FQAN) for example $foo.bar.foobar.ans$. Plans are in place for extending Jackal's solution to include Uniform Resource Locator (URL) based naming and addressing.

Jackal's naming scheme is based on part of the concept of localized name space defined in the SDSI/SPKI proposal for simple public key certificate [12] SDSI 1.0 localized name spaces solved the problems inherent in global name spaces as existed in the X.500 and X.509 global world-wide directory. Localized name spaces allows an agent to have different names according to the role they play while cooperating with other agents.

With this background, we must define exactly what constitutes an "agent identity" and how to bind this identity to an agent's name.

Based on the efforts of the teams identified with SPKI [7], SDSI [12], DTM [6], and DNSSEC [10], we propose identifying agents by their public keys. An agent represents a "principal." A principal, as defined in the literature [12, 7, 6, 10], is "an entity that supplies a service or requests an action in a distributed computing environment." As stipulated by the SDSI/SPKI proposal, agents speak by signing statements. Agents as principals will be considered the keyholders of the private (secret) key. Agents sign with their private key, thus the role of the public-key is one of signature verification.

In the following sections, we lay out the groundwork for agent security by defining the following: *Security Server Agent*, new performatives and new parameters needed to implement the security functions identified earlier, SSBL propositional content language, and finally trust management protocols associated with the security performatives.

3.2 Security Server Agent

We propose the following architecture in which a special agent named *Security Server Agent* (SSA) will be responsible for distributing certificates and other signed statements on behalf of the principal agent. See Figure 1. This SSA is considered a *Verifier* using SDSI/SPKI term for the trust computation engine, which is the entity which processes certificates, together with its own access control list entries, to determine if another agent deserves access or if some signed documents are valid. We proposed a one-to-one mapping between agents in SKQML and their SSA servers, this will make the SSA part verifier as well as part prover whenever its crossponding agent is participating in a trust management decision. This SSA could be part of the intra-agent composition. In other words, a KQML speaking agent will have sub-agents (threads) that are responsible for maintaining a local name space directory if it chooses to do so; or it could defer all the directory services to specialized agents (Facilitators, Brokers, or even specialized Agent Name Server agents). The KQML speaking agent can choose to do its own trust management system or it could defer that to a specialized SSA agent. To avoid the potential explosion of the number of agents, a group of agents from a particular domain could share one SSA for all their trust management functions. These agents must provide the shared SSA with their secret-keys, their access control list entries, and other authorization tags in order for the SSA to participate in trust management decisions on behalf of these agents.

In the following sections, we define details of the proposed extensions that are needed in order to meet the security functional requirements identified earlier. These details include the new KQML performatives (actions in another ACL) and parameters, SPKI-SDSI-based language for trust management as well as ontology denoting the meaning of the symbols in the content expression, and finally a number of protocols that help in the interpretation of the messages that are part of a conversation between SKQML speaking agents.

4 New KQML Performatives and Parameters

This section defines the individual message type that are needed to extend KQML with constructs (performa-

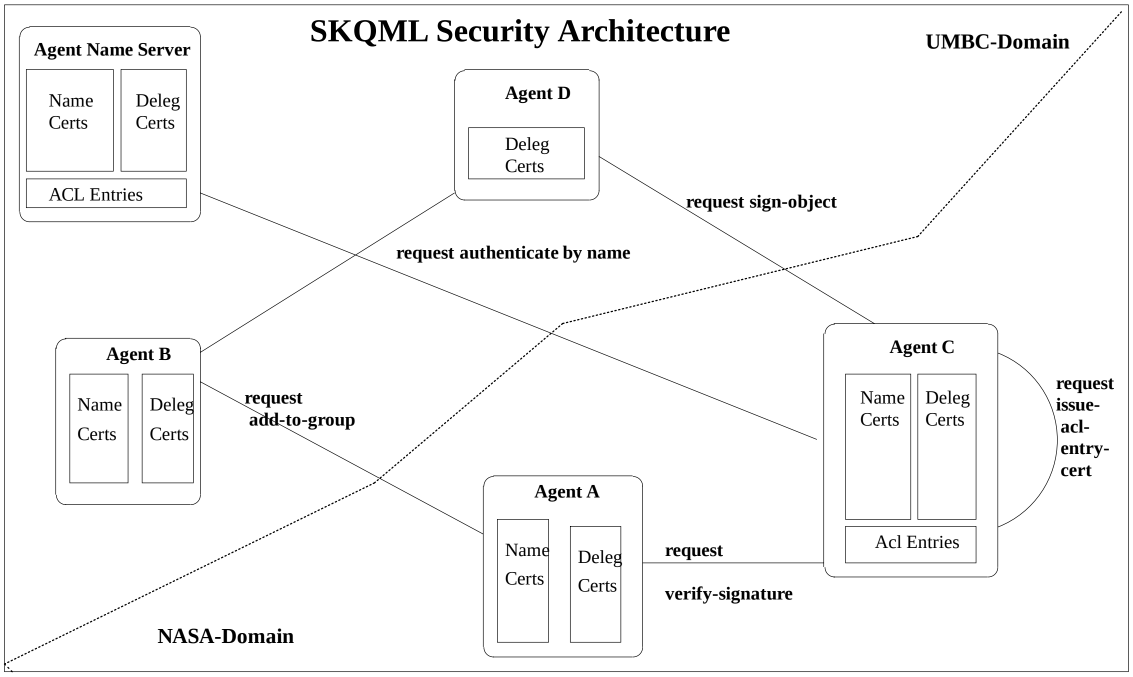

Figure 1. Overview of the SKQML Security Architecture

tives and parameters) to enable security and trust management. We start by reviewing some of the KQML message syntax and semantics. A KQML message [13, 14, 3] is expressed as an ASCII string using $s-expression$ language. Each KQML message starts with a speech act or performative (such as that it is a query, an assertion, a command, or any of a set of known performatives). The KQML messages are human readable, simple to parse, and easy to transport.

KQML performatives has parameters that are indexed by keywords. These parameters must begin with a (:) and must precede the corresponding value. For a complete semantic of the KQML messages, we refer the reader to [13, 14, 3].

4.1 Message Parameters

A KQML message has a set of well defined parameters that may occur in any order. Table 1 contains a review of current KQML parameters as well as description of the new parameters added as part of the SKQML proposal.

The only required parameter is the `:receiver` parameter. The expression associated with the `:content` parameter is what is being communicated to the receiver. The content can be encoded in any language specified in the `:language` parameter. The syntax of the SDSI-SPKI-based language is described in Section 5. Based on acts (performatives) introduced in FIPA ACL proposal part 2[5], we propose the following performatives to be part of the extended KQML or the SKQML.

4.2 Request Performative

The `request` performative pragmatics is that the sending agent requests that the receiving agent to perform some action described in the `content` parameter and specified in the `:language` parameter. The request performative can be used with the proposed SSBL, see Section 5, or it could be used with any other content language. As noted in the FIPA ACL standard document, the `request` act could be used to build composite conversations between agents by having the actions that are included in the content of the request to be themselves communicative acts.

It is worth noting that the new performative `request` is different than the KQML performative `achieve` in a number of ways. First, the meaning of `achieve` performative is that the sender would like the receiver to make something true of its environment while the in `request` performative the sender is requesting the receiver to execute or do a specific action. Second, it is true that one could argue that the `achieve` performative could be used in place of the `request` performative but that would require changing the semantic of the `achieve` to state explicitly that regardless of whether the content language is manipulative, declarative, or procedural.

Through out this paper, we use a university setting as a demonstration environment. All examples of communicating agents will be based on that environment. We assume the existence of software agents that speak SKQML for the following subset as a sample representative of possible agents in any university setting: Registration, Account-Payable, Administration, and Student-Agents. Each one of these agents is uniquely identified by its public-key. Later in 5, we shall explain how agents can send messages with `request` to themselves to generate their own keys, create their own access control list entries, generate auto-certificates, and many other functions that are needed for agents to participate in carrying out university related secure actions and secure functions.

Example: Agent Registration request agent Accounts-Payable to validate that student *Sam George* has no outstanding balance by providing AccountsPayable with the certificate that student *Sam George* supplied with his registration form as evidence of his eligibility to register. See Figure 2. The sender agent requests that the receiving agent employ a *MostCooperative* protocol; in other words, it is asking the receiving agent to try to get all required certificates in its effort to resolve this trust management request before responding to the sender with a list of missing certificates.

4.3 Refuse Performative

Agents sending the performative `refuse` are informing their recipients that the sending agent refuses to perform the action that has been requested by the receiver earlier. The sending agent attach an explanation for the refusal. The action to perform is described in the `content` parameter and specified in the `:language` parameter as well. See Figure 3 for full description of this performative.

Example: Agent *AccountsPayable* sends a `refuse` message to agent *Registration* in reply to a `request` that was sent earlier. See example in Figure 2. AccountsPayable explains the reason for refusal as `(insufficient-authorization-proofs validity-receipt-missing)` which included in the value of the `:content` parameter.Note that in the *refuse example* described in Figure 3, if the value for the `:protocol` parameter was *SemiCooperative* or *Most-Cooperative*, the receiving agent would have included a tag with the required certificates in the first protocol and the same tag with a list containing the remaining certificate that the receiving agent tried to get and failed thus requesting that the sender try to get those missing certificates by itself. The `result` of performing the action (or not in this case) is included as a sequence of certificates and signatures of tags. Also included is the action that was requested earlier. One could rely on the `:in-reply-with` field, but for the sake of completeness of the response, we recommend including the requested action as well.

4.4 Failure Performative

The *failure* performative is included so that the sending agent can inform the receiving agent that the request that the receiving agent had requested earlier failed and the reason for failure is included in the expression assigned to the `content` parameter using the specified `:language`. This performative in different than the *tell* performative since the sending agent doesn't require that the receiving agent to modify it belief thoughts or its Virtual Knowledge Base (VKB). See Figure 4 for full description of this performative.

Example: Agent *Registration* request agent *Accounts-Payable* to validate that student *Sam George* has no outstanding balance by providing *AccountsPayable* with the certificate that student *Sam George* supplied with his registration form as evidence of his eligibility to register. Agent *AccountsPayable* tries to verify the authenticity of the certificate and fails due to database error in its internal database and it is sending a `failure` message to inform the *Registration* agent with this information.

5 SDSI-SPKI-Based Language (SSBL) and Ontology

We define the SDSI-SPKI-Based Language (SSBL) propositional content language that enables representation of SDSI/SPKI actions, determination of the results of execution of these actions, completion of an action, representation of simple binary propositions, and introduces boolean connectives to represent propositional expressions. Figure 5 describes the SSBL grammar in BNF form.

The definition of the `SDSPExpr` terms is defined in [7].

5.1 Pragmatics of the SSL Language

There are two types of actions, *Intra-Agent* and *Inter-Agent*, Intra-agent actions are those actions requested using the `request` performative and sent to by an agent to itself to initialize its name certificate database, generate public-key pair, generate auto-certificates, and generate delegation certificate. The Inter-agent actions are those requested using the `request` performative where the action included in the `:content` is to be performed by the trust management engine of the `:receiver` agent and the result of execution returned to the sender agent either via a `tell` performative, `failure`, or `deny`.

Inter-Agent Actions:

- *register-agent* The sender is registering itself with the Agent Name Server (ANS) agent; or any other agent capable of holding *name certificates*, assigned to the receiver field. The process of registering an agent with ANS start by sending a request performative with the parameter content field that contains the action *register-agent* and either the $s - expression$ containing the *auto-certificate* or a name certificate with Public-Key as an S-expression that represents a SDSI-SPKI public key object as well as the name that the registering agent would like to assume.

  In the example described in Figure 6, agent *SamGeorgeAgent* is acting on behalf of student *Sam George* and is trying to add its public-key as well as any related information to the localized name space of the *Registration* agent.

- *authenticate-agent-by-name* The sending agent request that the receiving agent verify that it has a valid name certificate that matches that certificate included in the content of the request message. The receiver object can respond with a `tell` message that has a `:content` value that contains a `result` SSBL construct with either *true* or *false*. In the example described in Figure 7, agent *Registration* asks agent *VerificationServer* which might be a public server where agents could register their names, to authenticate that *Sam George* public key is bound to that name in their certificate database.

- *authenticate-agent-by-key* The sending agent request that the receiving agent verify that it has a valid a certificate that matches that certificate included in the content of the request message. This certificate includes the $public - key$ SDSI object. The receiver object can respond with a `tell` message that has a `:content` value that contains a `result` SSBL construct with either *true* or *false*.

- *sign-object* The sender object requests that the receiver signs the enclosed object with the receiver public-key. It will return a SDSI sequence object with the signature object of the supplied object. It will return the SDSI sequence object via a `result` SSL construct contained within the body of the content parameter of a `tell` performative.

- *hash-object*

  The sender object requests that the receiver object hash the enclosed object using the hash algorithm mentioned in the content message. It will return the hashed object via a `result` SSBL construct contained within the body of the content parameter of a `tell` performative.

- *check-authorization* The sender object request that the receiver object check the validity of an authorization certificate. The receiver object can respond with a `tell` message that has a `:content` value that contains a `result` SSBL construct with either *true* or *false* or it can respond with a result that contains the certificate that the receiver holds signed with the receiver's own key.

- *check-membership* The sender object requests that the receiver object checks that an agent is a member of a particular group. The receiver object can respond with a `tell` message that has a `:content` value that contains a `result` SSL construct with either *true* or *false* or it can respond with a result that contains the certificate that the receiver holds signed with the receiver's own key.

- *verify-signature* The sender object requests that the receiver object checks that the signature included is a valid one. The receiver object can respond with a `tell` message that has a `:content` value that contains a `result` SSBL construct with either *true* or *false* or it can respond with a result that contains the certificate that the receiver holds signed with the receiver's own key.

- *list-required-cert* The sender object requests that the receiver object returns a list of required certificates. The receiver object can respond with a `tell` message that has a `:content` value that contains a `result` SSL construct with a sequence of tag certificate (not signed of course).

- *add-to-group* The sender object requests that the receiver object adds the sender's name to the group identified by the name certificate included in the content of the request message. The receiver object can respond with a `tell` message that has a `:content` value that contains a `result` SSL construct with either *true* or *false* in case of success of failure respectively, or it can respond with a result that contains the certificate that the receiver created as a result of the addition signed with the receiver's own key.

- *reconfirm* The sender object requests that the receiver agent re-confirm the validity of the certificate included in the body of the content of the request message. Either *true* or *false* in case of success of failure respectively, or it can respond with a result that contains the certificate that the receiver created as a result of the addition signed with the receiver's own key.

Intra-Agent Actions:
All messages are sent within the same sender agent. The reason we allow this so implementation interpreters of the SSL language can use the same set of Application Programming Interfaces (APIs) for Inter-agent as well as Intra-agent.

- *generate-key* An agent will send a message to itself with a `self-action` construct of the SSL Language to generate a pair of public and private keys. The private key will be used to sign message and will remain private (hidden) in its own memory. The corresponding public part will be available upon request by other agents, so it will be used in signing (encrypting) messages sent to the holder of the secret key that matches this public-key.
  The generate-key action takes a SDSI-SPKI tag argument that specifies the generating method. In our example we used the tag *pgp* with a key length of 1024. See Figure 8.

- *issue-auto-cert*
  An agent can use this `self-action` to generate an auto-certificate that includes whatever the agent would like the rest of the world agents to know about itself.

- *issue-local-name-cert* An agent can use this `self-action` to issue a local name certificate. This certificate will be stored in the agents *name certificates* database.
  In the following example, agent *CSEE.Chairperson* defines a local name entry in his localized name space for the *CS.Authorization.Agent* and he named it `CS Authorization Agent`.

- *issue-acl-entry-cert* An agent can use this `self-action` to issue an *acl* entry that will be stored in an *acl certificates* database.

- *issue-delg-cert* An agent can use this `self-action` to an *delegation* certificate that will be stored in an *delegation certificates* database.

- *issue-group-member-cert* An agent can use this `self-action` to a *group* certificate which is a kind of a *name* certificate.

- *encrypt-object* An agent can use this `self-action` to encrypt an object with its own key. The agent encrypts the content of the object(s) mentioned in the content message. It will return the encrypted object via a `result` SSL construct contained within the body of the content parameter of a `tell` performative.

- *decrypt-object* An agent can use this `self-action` to decrypt an encrypted object with its own key.The agent decrypts the content of the object included with the encryption key given the content message. It will return the decrypted object via a `result` SSL construct contained within the body of the content parameter of a `tell` performative.

We assume that KQML-speaking agents will use a basic agent ontology, which provides a small set of classes, attributes, and relations. The major assumption in our ontology is that it burrows most of the definitions of its classes, attributes, and relations from the SDSI/SPKI data structures and syntax.

- Principal is a term that refers to a signature key. Or he private part of the public-key. It is used to sign messages on behalf of the agent which this agent represents.
- Public-key <SDSI Public-Key Object> this term refers a SDSI public key object which will be used to verify the signatures of certificates signed by the principal agent bound to this key.
- Key-holder is an entity that holds the secret key. It is the agent that is holding that key or the human that this agent is representing.
- Name is a string of the form $K_1 N_1 N_2 \cdots N_k$ where $1 \leq k$. Every name is part of a name space which is localized to the agent that holds the name certificate for that name.
- Certificate is a digitally signed record containing name and a public key.
- Name Certificate is a certificate that binds a name to a principal or a group of principals.
- Authorization Certificate is a certificate that binds an authorization to a principal or a group of principals.

6 Protocols for Trust Management

A protocol is a set of actions and responses to those actions that must be fulfilled in order to be compliant with that protocol. We define a number of protocols for trust management of certificates. Each protocol describes what is expected from an agent while responding to a *request* performative.

- **Cooperative** Agents agree to a minimal cooperation whenever trust management issues are involved. Meaning that an agent is not going to go out of its way to gather the needed certificates or for that matter to volunteer the information about those needed certificates to the requesting agent.
- **SemiCooperative** The sending agent requests that the receiving agent to be a little bit more cooperative than that in the *Cooperative* protocol case. Agents might inform others with the kind of certificate required to carry through the trust management issues being discussed among them.
- **MostCooperative** The sending agent is requesting that the receiving agent try to retrieve whatever they deem necessary to carry out the trust management issues discussed among them. The receiving agent can respond with `deny` performative if it sees that this kind of cooperation involved is beyond the kind of actions that it can perform to carry its trust management obligations.

7 Conclusion

The proposed SKQML fixed the lack of security constructs in the agent communication languages standards by providing an infrastructure for security that is based on open cryptographic certificate standards. This will guarantee interoperability as well as ease of integration with existing and yet to be implemented trust management engines. Our proposal allows agents to participate in trust management issues at a level that is appropriate for meaningful interactions among agents.

In conclusion, SKQML is simple, extensible, at a level appropriate for intelligent agents, requires the addition of very few new performatives, is based on public-key cryptographic standards, and provides security functions as an integral part of the communication language.

Acknowledgments

We thank Alan Sherman for helpful editorial comments.

References

[1] Tim Finin, Rich Fritzon, Don McKay, and Robin McEntire. KQML – A language and protocol for knowledge and information exchange. In *Proceedings of the 13th International Workshop on*

Distributed Artificial Intelligence, pages 126–136, Seatle, WA, July 1994.

[2] Tim Finin, Rich Fritzson, Don McKay, and Robin McEntire. KQML - A language and protocol for knowledge and information exchange. Technical Report CS-94-02, Computer Science Department, University of Maryland and Valley Forge Engineering Center, Unisys Corporation, Computer Science Department, University of Maryland, UMBC Baltimore MD 21228, 1994.

[3] The DARPA Knowledge Sharing Initiative External Interfaces Working Group. Specification of the KQML agent-communication language (draft version), 1993.

[4] J. Mayfield, Y. Labrou, and T. Finin. Evaluating KQML as an agent communication language. In M. Wooldridge, J. P. Müller, and M. Tambe, editors, *Intelligent Agents II (LNAI 1037)*, pages 347–360. Springer-Verlag: Heidelberg, Germany, 1996.

[5] Foundation For Inelligent Physical Agents. FIPA 97 Specification, Part 2, Agent Communication Languages, 1997.

[6] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 1996.

[7] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B.M. Thomaa, and T. Ylonen. Simple public key certificate, internet-draft, 1997.

[8] Chelliah Thirunavukkarasu, Tim Finin, and James Mayfield. Secret agents - a security architecture for KQML. In Tim Finin and James Mayfield, editors, *Proceedings of the CIKM '95 Workshop on Intelligent Information Agents*, Baltimore, Maryland, 1995.

[9] Chelliah Thirunavukkarasu, Tim Finin, Don McKay, and Robin McEntire. On agent domains, agent names andd proxy agents. In Tim Finin and James Mayfield, editors, *Proceedings of the CIKM '95 Workshop on Intelligent Information Agents Workshop*, Baltimore, Maryland, 1995.

[10] D. Eastlake and C. Kaufman. Domain name system secutiry extensions, 1997.

[11] R. Scott Cost et. la. Jackal: A JAVA implemenation of KQML. Web Pages URL: http://jackal.cs.umbc.edu/∼cost/J3.

[12] Butler Lampson and Ron Rivest. SDSI - A Simple Distributed Security Infrastructure, 1996.

[13] Yannis Labrou and Tim Finin. A proposal for a new KQML specification. Technical report, University of Maryland Baltimore County, 1997.

[14] Yannis Labrou. *Semantics for an agent communication language*. PhD thesis, Unviersity of Maryland Graduate School, 1996.

Message Parameter	**Meaning**
`:sender`	denotes the identity of the sender of the message this identity could be the name of the agent using localized name space of fully qualified name supplied by the ANS. This name denotes the name of the agent sending the message.
`:receiver`	denotes the identity or identities or the recipient of the message. Multiple agent names can be included in an *n-tuple*. This notion of multicast doesn' t exit in the pragmatic of the current KQML but it does exit in the FIPA ACL proposal.[5]
`:from`	the origin of the performative in `:content` when the *forward* is used.
`:to`	the final destination of the performative in `:content` when the *forward* is used.
`:reply-with`	introduces the expected label (expression) which will be used in response to the current message. This label can be used to follow up on current or previous conversations.
`:reply-by`	denotes the time and/or date which indicates the latest time/date by which the sender expects a reply. This is a new parameter to be added to the reserved set of parameters of any KQML message.
`:in-reply-to`	denotes the expected label (expression) in response to a previous action to which this message is a reply.
`:language`	denotes the name of the representation language of the `:content` parameter for the action of the current message.
`:ontology`	denotes the name of the ontology which is used to give term definitions for the symbols used in the `:content` parameter.
`:protocol`	introduces an identifier denoting the protocol which the sender is employing. This protocol name will aid the receiver in interpreting the `:content` parameter expression. For example, a protocol to help establish the level of cooperation the sender of a `request` performative is expecting from the receiver while processing security related certificates.
`:conversation-id`	a label that can be used as aid in an on-going conversation between communicating agents. This label could also aid in the interpretation of the `:content` parameter expression.

Table 1. Summary of reserved KQML parameters and their meanings.

request	
Summary	the sending agent requests that the receiving agent to perform some action described in the `content` parameter and specified in the `:language` parameter.
Message content	expression containing the action to be performed
Description	the sending agent requests that the receiving agent to perform some action described in the `content` parameter and specified in the `:language` parameter. The receiver can do one of the following: • choose to accept to perform the action and inform the sender with the results of the execution of the action by sending a *tell* performative in case of success or sending a `failure` performative (See details of `failure` performative in Table 4) in case the attempt to execute the action ended in failure. Note that the `failure` message will contain an explanation for what happened in the `:content` parameter. • choose to refuse to perform the action by sending a `refuse` performative explaining the reason for refusal. See details of `refuse` performative in Table 3. SDSI-SPKI-Lang will be used to build the content expressions.

Table 2. Request Performative Definition

refuse	
Summary	the sending agent refuses to perform an action that has been requested by the receiver earlier. The sending agent attach an explanation for the refusal. The action to perform is described in the `content` parameter and specified in the `:language` parameter.
Message content	a sequence of s-expressions to describe both the action requested as well as a proposition describing the reasons for refusal.
Description	the sending agent refuses to perform the action requested as part of the message content of an earlier message with the *request* performative as the action being communicated. The sending agent of the refuse act is entitled to deny the execution of any request made from any other agent. The receiving agent of a `refuse` message can interpret this message as either the action has not been done or the action is not feasible from the point of view of the sender or the reason for the refusal as presented in the content of the message being sent.

Table 3. Refuse Performative Definition

```
(request
        :sender   Registration
        :receiver AccountsPayable
        :reply-with validity-check1
        :language SDSI-SPKI-Lang
        :ontology SDSI-SPKI-Ontology
        :protocol MostCooperative
        :content
           (action AccountsPayable
             (check-authorization
               (sequence
                (cert
                  (issuer (hash md5 |YIojiXGq2xd1eZzt+bpYQg==|))
                  (subject (name
                  (hash md5
                   |HnI4+GLQRWgj/sB8IgT1Cw==|) Sam George))
                  (tag (eligible-to-register))
                )

               (signature
                 (hash md5 |iAbKf5zthRC5muyT/uCdWg==|)
                 (public-key
                  rsa-pkcs1-md5
                  (e #11#)
                  (n
                   |AKBKJPG49slRYDpAs2hGACjPcg4b9STLjixg1HedxMI
                   AqI2a3dB/BVOgBbHhDX/aNWfJdo7Hv2caV6DoU+9/Uik=|))
                   |VazQKWIA488H+s3xOqOj+G/hr2/leHJI0yhmK8Y4MQrJy2
                   STMIwMq5PrHhAHgNxc36nfcv6u/DhnfP9a3KnvWQ==|)
                )
             )
          )
```

Figure 2. Request Performative Example

```
(refuse
        :sender AccountPayable
        :receiver Registration
        :in-reply-to validity-check1
        :language SDSI-SPKI-Lang
        :ontology SDSI-SPKI-Ontology
        :protocol Cooperative
        :content
        (result
           (action AccountsPayable
             (check-authorization
               (sequence
                (cert
                  (issuer (hash md5 |YIojiXGq2xd1eZzt+bpYQg==|))
                  (subject (name
                  (hash md5
                   |HnI4+GLQRWgj/sB8IgT1Cw==|) Sam George))
                 (tag (eligible-to-register)))
               (signature
                 (hash md5 |iAbKf5zthRC5muyT/uCdWg==|)
                 (public-key
                  rsa-pkcs1-md5
                  (e #11#)  (n
                   |AKBKJPG49slRYDpAs2hGACjPcg4b9STLjixg1HedxMI
                   AqI2a3dB/BVOgBbHhDX/aNWfJdo7Hv2caV6DoU+9/Uik=|))
                   |VazQKWIA488H+s3xOqOj+G/hr2/leHJI0yhmK8Y4MQrJy2
                   STMIwMq5PrHhAHgNxc36nfcv6u/DhnfP9a3KnvWQ==|))))

           (sequence
            (cert
            (issuer  (hash md5 |YIojiXGq2xd1eZzt+bpYQg==|))
            (subject  (public-key    rsa-pkcs1-md5
            (e #11#)  (n
            |APvZ9UAXPUM/tYHYnoCuXUjJUN4fTh/SANGh/UwCPLbtcK
            vTrA9H1NV+CMGTuj4pps4F0dDm6ZzywAJEwH0QbX0=|)))
            (tag (reason-for-refusal
                (insufficient-authorization-proof
                                      validity-missing))))
            (signature
            (hash md5 |6mF5rWWdbZKN3qYfP+5+eA==|)
            (public-key rsa-pkcs1-md5
           (e #11#)      (n
           |AKBKJPG49slRYDpAs2hGACjPcg4b9STLjixg1HedxMIAq
           I2a3dB/BVOgBbHhDX/aNWfJdo7Hv2caV6DoU+9/Uik=|))
           |Zqx/yWMI4MWVRcFldAPmYiC9losZKD135wj/X6PRTonVY
           Clpsn5OIeB8Z18kIhWMudV2itPtynyooK2ziZklqg==|)) )
```

Figure 3. Refuse Performative Example

failure	
Summary	the sending agent informs the receiving agent that the request that the receiving agent had requested earlier failed and the reason for failure is included in the expression assigned to the `content` parameter using the specified `:language`.
Message content	expression containing tuple of the action to be performed and an s-expression explaining the reason for failure.
Description	the sending agent requests that the receiving agent to be informed that the action described in an s-expression in `content` parameter and specified in the `:language` parameter did fail. The receiver can do choose to believe one of the following: • the action requested earlier has not been done. • the sender tried and failed to perform the action and the sender is by sending a `failure` performative explaining the reason for refusal. SDSI-SPKI-Lang will be used to build the content expressions.

Table 4. Failure Performative Definition

```
(failure
        :sender   AccountsPayable
        :receiver Registration
        :in-reply-to validity-check1
        :language SDSI-SPKI-Lang
        :ontology SDSI-SPKI-Ontology
        :protocol Cooperative
        :content
        (result
           (action AccountsPayable
             (check-authorization
               (sequence
                (cert
                  (issuer (hash md5 |YIojiXGq2xd1eZzt+bpYQg==|))
                  (subject (name
                  (hash md5
                   |HnI4+GLQRWgj/sB8IgT1Cw==|) Sam George))
                 (tag (eligible-to-register)))
               (signature
                 (hash md5 |iAbKf5zthRC5muyT/uCdWg==|)
                 (public-key    rsa-pkcs1-md5
                  (e #11#)  (n
                   |AKBKJPG49slRYDpAs2hGACjPcg4b9STLjixg1HedxMI
                   AqI2a3dB/BVOgBbHhDX/aNWfJdo7Hv2caV6DoU+9/Uik=|))
                   |VazQKWIA488H+s3xOqOj+G/hr2/leHJI0yhmK8Y4MQrJy2
                   STMIwMq5PrHhAHgNxc36nfcv6u/DhnfP9a3KnvWQ==|))) )
        (sequence
        (cert
        (issuer  (hash md5 |YIojiXGq2xd1eZzt+bpYQg==|))
        (subject  (public-key    rsa-pkcs1-md5
        (e #11#)    (n
        |APvZ9UAXPUM/tYHYnoCuXUjJUN4fTh/SANGh/UwCPLbtc
        KvTrA9H1NV+CMGTuj4pps4F0dDm6ZzywAJEwH0QbX0=|)))
        (tag (reason-for-failure
                 (internal-database-error read-error))))
        (signature
           (hash md5 |jqb7l4Rn+FpMEW5mfIXVkA==|)
           (public-key
             rsa-pkcs1-md5
             (e #11#)
             (n
             |AKBKJPG49slRYDpAs2hGACjPcg4b9STLjixg1HedxMIAq
             I2a3dB/BVOgBbHhDX/aNWfJdo7Hv2caV6DoU+9/Uik=|))
             |AJJTlkBjctOjbXRZEgYkgU/KaIDil4FCN7XPEe9i0Tpy8l
             fMLKvgeNJskTfe/z50nRvhSKeD6sTyIeph1PAHBnI=|))
       )
```

Figure 4. Failure Performative Example

```
SSBLContentExpr ::=   SSBLExpr
                 |  SSBLActionExpr.
SSLBExpr        ::=   SSBLFormula
                 | ``(`` ``not''        SSBLExpr ``)''
                 | ``(`` ``and''        SSBLExpr SSBLExpr ``)''
                 | ``(`` ``or'' SSLExpr SSBLExpr ``)''
                 | ``(`` ``implies''   SSBLExpr SSBLExpr ``)''
                 | ``(`` ``equiv''     SSBLExpr SSBLExpr ``)''.
SSBLFomula      ::=  ``(`` ``result'' SSBLActionExpr SSBLTerm ``)''
                 | ``true''
                 | ``false''
                 | ``undecided''.
SSBLTerm        ::=   SDSPExpr
                 | SSBLFuncTerm
                 | SSBLActionExpr.
SSBLActionExpr  ::=  ``(`` ``action'' SSBLAgent SSLFuncTerm ``)''
                 | ``(`` ``self-action'' SSBLSFuncTerm ``)''.
SSBLFuncTerm    ::=   ``(`` SSBLFuncSymbol      SSBLTerm* ``)''.
SSBLSFuncTerm   ::=   ``(`` SSBLSFuncSymbol     SSBLTerm* ``)''.
SSBLAgent       ::=   AgentName.
SSBLFuncSymbol  ::=  ``authenticate-agent-by-name''
                 | ``authenticate-agent-by-key''
                 | ``sign-object''
                 | ``hash-object''
                 | ``check-authorization''
                 | ``check-membership''
                 | ``verify-signature''
                 | ``list-required-cert''
                 | ``add-to-group''
                 | ``register-agent''
                 | ``reconfirm''


SSBLSFuncSymbol ::=  ``generate-key''
                 | ``issue-auto-cert''
                 | ``issue-local-name-cert''
                 | ``issue-acl-entry-cert''
                 | ``issue-delg-cert''
                 | ``issue-group-member-cert''
                 | ``encrypt-object''
                 | ``decrypt-object''.

SDSPExpr        ::=  <5-tuple>
                 | <acl>
                 | <crl> | <delta-crl> | <reval>
                 | <sequence>.
```

Figure 5. SSL BNF

```
(request
:sender   SamGeorgeAgent
:receiver Registration
:reply-with validity-check1
:language SDSI-SPKI-Lang
:ontology SDSI-SPKI-Ontology
:protocol MostCooperative
:content
  (action SamGeorgeAgent
    (register-agent
      (name
      (public-key
      rsa-pkcs1-md5
      (e #11#)
      (n
  |AJ24Vi1To6SMzm76GeGB16fBm330qV9xDe/zkjgY1Ir7nAUXuJEj0ic7
  J18O8TEE6bSWKjyLHeeivquXnGYV8A0=|))        Sam George)

   (signature
   (hash md5 |3yZ51jNZx70YnNSLwqQlCw==|)
   (public-key
    rsa-pkcs1-md5
    (e #11#)
    (n
 |AKlB6EvdWXqsO5myvSjSiLYw3rQlVOIdoQnX6rXlRjUvzJqWZH26qsk8
 GLLdchRD0L5qGwZDsEsBSp07xF6jCsE=|))
 |AIiCAm0zpQtjF5MpHdCMWjovUHGg3rzzjnn8PgCK7bVhFRT4LV33I48mNi
 YHfaQkCY3vSoMthfyXDQ5RSZjfiZU=|)
    )
  )
 )
```

Figure 6. Register-agent Action Example

```
(request
:sender   Registration
:receiver VerificationServer
:reply-with validity-check1
:language SDSI-SPKI-Lang
:ontology SDSI-SPKI-Ontology
:protocol MostCooperative
:content
  (action Registration
   (authenticate-agent-by-name
    (name
      (public-key
       rsa-pkcs1-md5
       (e #11#)
       (n
  |AJ24Vi1To6SMzm76GeGB16fBm330qV9xDe/zkjgY1Ir7nAUXuJEj0ic7
  J18O8TEE6bSWKjyLHeeivquXnGYV8A0=|))           Sam George)
   )
 )
```

Figure 7. Authenticate-agent-by-name Action Example

```
(request
:sender   Dr.John.Doe
:receiver Dr.John.Doe
:reply-with  autocert1
:language SDSI-SPKI-Lang
:ontology SDSI-SPKI-Ontology
:protocol MostCooperative
:content
 (self-action Dr.John.Doe
  (generate-key
   (tag (pgp 1024)))
   )
  )
)
```

Figure 8. Generate-key Example