# REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188



| 1. AGENCY USE ONLY (Leave blank) | 2. REPORT DATE | 3. REPORT TYPE AND DATES COVERED |
|---|---|---|
| | | FINAL 17 AUG 93 TO 16 AUG 95 |

**4. TITLE AND SUBTITLE**

AASERT92- FACILITATORS AND MEDIATORS FOR INTILLIGENT AGENT PROTOCOLS

**5. FUNDING NUMBERS**

F49620-93-1-0506
3484/ZS 61103D

**6. AUTHOR(S)**

TIMOTHY W. FININ

AFOSR-TR-96

0381

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**

COMPUTER SCIENCE & ELECTRICAL ENGINEERING
UNIVERISTY OF MARYLAND BALTIMORE COUNTY
5401 WILKINS AVE
BALTIMORE MD 21228

**9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)**

AFOSR/NM
110 DUNCAN AVE, SUITE B115
BOLLING AFB DC 20332-8080

**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**

F49620-93-1-0506

**11. SUPPLEMENTARY NOTES**

**12a. DISTRIBUTION/AVAILABILITY STATEMENT**

APPROVED FOR PUBLIC RELEASE
DISTRIBUTION UNLIMITED

**12b. DISTRIBUTION CODE**

**13. ABSTRACT (Maximum 200 words)**

SEE REPORT FOR ABSTRACT

19960726 037

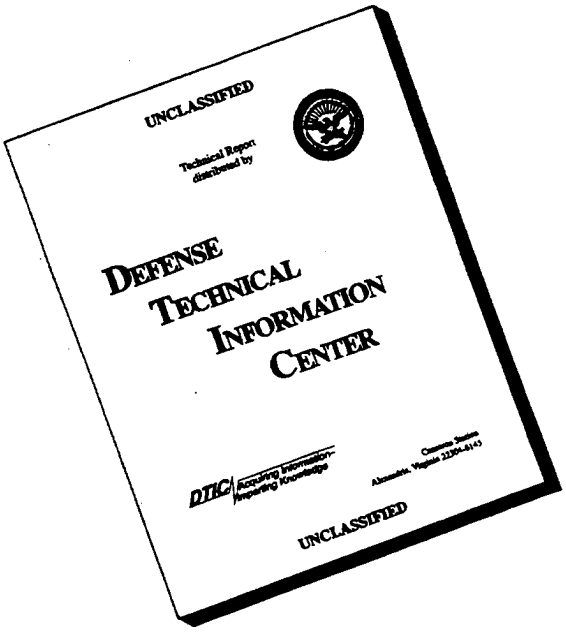
| 14. SUBJECT TERMS | | | 15. NUMBER OF PAGES |
|---|---|---|---|
| | | | 16. PRICE CODE |
| 17. SECURITY CLASSIFICATION OF REPORT | 18. SECURITY CLASSIFICATION OF THIS PAGE | 19. SECURITY CLASSIFICATION OF ABSTRACT | 20. LIMITATION OF ABSTRACT |
| UNCLASSIFIED | UNCLASSIFIED | UNCLASSIFIED | SAR |

NSN 7540-01-280-5500

DTIC QUALITY INSPECTED 1

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18
298-102

# DISCLAIMER NOTICE




**THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.**

Final

R/A 96/05/14

Technical Report

# Facilitators and Mediators for Intelligent Agent Protocols

## Security and Agent Names

October 1995

Timothy W. Finin
Computer Science and Electrical Engineering
University of Maryland Baltimore County
5401 Wilkins Avenue
Baltimore MD 21228
finin@umbc.edu

Project F49620-93-1-0506
Sponsored by
Air Force Office of Scientific Research
Bolling AFB DC, 20332-6448
AASERT Program
Augementation Awards for Science
and Engineering Research Training

# 1 Introduction

This project focuses on the development of and experimentation with a a language and protocol intended to support interoperability among intelligent agents in a distributed application. Examples of applications envisioned include intelligent multi-agent design systems supporting collaborative designs of complex circuits and devices by multiple design teams as well as intelligent planning, scheduling and replanning agents supporting distributed transportation planning and scheduling applications. The language, KQML for Knowledge Query and Manipulation Language, is part of a larger DARPA-sponsored Knowledge Sharing effort focused on developing techniques and tools to promote the sharing on knowledge in intelligent systems (Neches 1991).

The technical problem of coordinating many agents who must communicate with one another is a difficult one. In general, the agents must know a lot:

- Which other agents to communicate with.
- How to establish a reliable communication channel with them.
- What protocol to use in the ensuing dialogue.
- What language to use to exchange information knowledge.
- What terms within the language to use to guarantee that the other agent will interpret the expressions in the same way.
- How to handle inconsistent information and the eventual mis-matches that arise from different views of the world.

These problems are compounded when we are trying to coordinate a multitude of "intelligent agents". The key aspects of the problem specifically addressed in this project are

- the definition of a standard KQML for use in current and future intelligent information integration projects,
- the development of an efficient, robust and scalable KQML implementation which we will call the Simple Knowledge Transfer Protocol (SKTP),
- theoretical and practical development of mediators for knowledge-based integration of information

The remainder of this report consists of two recent papers which discusses various aspects of KQML.

The first, *"A Security Architecture for the KQML Agent Communication Language"*, by Chelliah Thirunavukkarasu , Tim Finin and James Mayfield discusses the security features that a KQML user would expect and an architecture to satisfy those expectations. The proposed architecture is based on cryptographic techniques and would allow agents to verify the identity of other agents, detect message integrity violations, protect confidential data, ensure non-repudiation of message origin and take counter measures against cipher attacks.

The second paper, *"On Agent Domains, Agent Names and Proxy Agents"* considers the problem of how software agents should be named and what kind of infrastructure is necessary in order to locate an agent given only its name. We assume an agent environment which (1) is dynamic with agents being created and destroyed frequently; (2) undergoes re-organizations with agent groups and sub-groups forming and disbanding; and (3) supports agent communication by any of several transport mechanisms such as TCP/IP, email, http and distributed object systems. This leads us to propose the establishment of agent domains which are organized into an agent domain hierarchy. Agent name resolution can be done by agent name servers, analogous to Internet domain name servers. One of the additional benefits from this approach is that it easily supports the definition of proxy agents. We sketch how this proposal would impact the KQML agent communication language and protocol and describe an ongoing implementation of a generic KQML Agent Name Server and its integration into the KATS framework.

# A Security Architecture for the KQML Agent Communication Language

Chelliah Thirunavukkarasu
Enterprise Integration Technologies
Menlo Park, CA 94025
chelliah@eit.com

Tim Finin and James Mayfield
Computer Science and Electrical Engineering
University of Maryland Baltimore County
Baltimore MD 21228 USA
finin@umbc.edu - mayfield@umbc.edu

**Address correspondence to:** Tim Finin, Computer Science and Electrical Engineering, University of Maryland Baltimore County, Baltimore MD 21228 USA, finin@umbc.edu, 410-455-3522, fax: 410-455-3969.

**Acknowledgements:** This work has been the result of very fruitful collaborations with a number of colleagues with whom we have worked on KQML and other aspects of the Knowledge Sharing Effort. We wish to specifically thank and acknowledge Don McKay, Robin McEntire, Richard Fritzson, Charles Nicholas, Yannis Labrou, R. Scott Cost, and Anupama Potlori. Arulnambi Kaliappan suggested the agent names and and Senthil Periaswamy of the University of South Carolina provided stimulating discussions on possible security threats and attacks. This work was supported in part by the Air Force Office of Scientific Research under contract F49620-92-J-0174, and the Advanced Research Projects Agency monitored under USAF contracts F30602-93-C-0177 and F30602-93-C-0028 by Rome Laboratory.

**Abstract:** Knowledge Query and Manipulation Language (KQML) is a communication language and protocol which enables autonomous and asynchronous agents to share their knowledge and or work towards cooperative problem solving. With the popularity of internet and the possibilities offered by the agent technology we can expect an explosion of agents in the near future. For an agent communication language like KQML to be effective in an open environment, it should provide some means for agents to communicate in a secure manner to protect the privacy and integrity of data and to provide for the authentication of other agents. In this paper we discuss some basic and extended security requirements for software agents and an architecture to satisfy those requirements for KQML-speaking agents. The proposed architecture is based on cryptographic techniques and would allow agents to verify the identity of other agents, detect message integrity violations, protect confidential data, ensure non-repudiation of message origin and take counter measures against cipher attacks.

Submitted to AAAI-96 with **tracking number A615** under **content areas enabling technologies, multi-agent systems, software agents** with approximately **6750 words**. This paper has not been submitted to any other conferences. An earlier version was included in the online proceedings of the CIKM Workshop on Intelligent Information Agents.

# A Security Architecture for the KQML Agent Communication Language

**Abstract:** Knowledge Query and Manipulation Language (KQML) is a communication language and protocol which enables autonomous and asynchronous agents to share their knowledge and or work towards cooperative problem solving. With the popularity of internet and the possibilities offered by the agent technology we can expect an explosion of agents in the near future. For an agent communication language like KQML to be effective in an open environment, it should provide some means for agents to communicate in a secure manner to protect the privacy and integrity of data and to provide for the authentication of other agents. In this paper we discuss some basic and extended security requirements for software agents and an architecture to satisfy those requirements for KQML-speaking agents. The proposed architecture is based on cryptographic techniques and would allow agents to verify the identity of other agents, detect message integrity violations, protect confidential data, ensure non-repudiation of message origin and take counter measures against cipher attacks.

Submitted to AAAI-96 with **tracking number A615** under **content areas enabling technologies, multi-agent systems, software agents** with approximately **6750 words**. This paper has not been submitted to any other conferences. An earlier version was included in the online proceedings of the CIKM Workshop on Intelligent Information Agents.

## 1 Introduction

Knowledge Query and Manipulation Language (KQML) [1] is a communication language and protocol which enables autonomous and asynchronous software agents to share their knowledge and or work towards cooperative problem solving. With the popularity of internet and the possibilities offered by the agent technology we can expect an explosion of agents in the near future. For an agent communication language like KQML to be effective in an open environment, it should provide some means for agents to communicate in a secure manner to protect the privacy and integrity of data and to provide for the authentication of other agents. In this paper we discuss some basic and extended security requirements for software agents and an architecture to satisfy those requirements for KQML-speaking agents. The proposed architecture is based on cryptographic techniques and would allow agents to verify the identity of other agents, detect message integrity violations, protect confidential data, ensure non-repudiation of message origin and take counter measures against cipher attacks.

### 1.1 Background on KQML

KQML is a high-level language intended for the run-time exchange of knowledge between intelligent systems. It was developed as a part of the *Knowledge Sharing Effort* [20, 19, 13]. The KQML language can be thought of as consisting of three layers: the content layer, the message layer, and the communication layer. The content layer bears the actual content of the message. in the programs own representation language. KQML can carry expressions encoded in any representation language, including languages expressed as ASCII strings and those expressed using a binary notation. Some KQML-speaking agents (e.g., routers, very general brokers, etc.) may ignore the content portion of the message, except to determine where it ends.

The communication level encodes a set of message features which describe the lower level communication parameters, such as the identity of the sender and recipient, and a unique identifier associated with the communication.

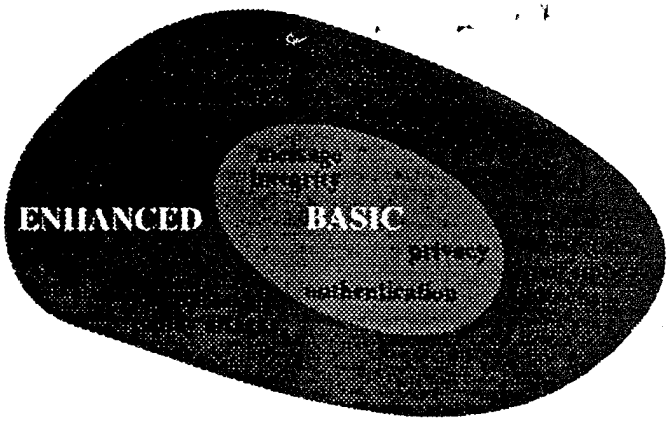



Figure 1: **We propose a security architecture for the KQML agent communication language with two levels – basic and enhanced – which offer different capabilities and have different costs associated with them.**

It is the message layer that is used to encode a message that one application would like to transmit to another. The message layer forms the core of the KQML language, and determines the kinds of interactions one can have with a KQML–speaking agent. A primary function of the message layer is to identify the protocol to be used to deliver the message and to supply a speech act or performative which the sender attaches to the content (such as that it is an assertion, a query, a command, or any of a set of known performatives). In addition, since the content may be opaque to a KQML-speaking agent, this layer also includes optional features which describe the content language, the ontology it assumes, and some type of description of the content, such as a descriptor naming a topic within the ontology. These features make it possible for KQML implementations to analyze, route and properly deliver messages even though their content is inaccessible.

The syntax of KQML is based on a balanced parenthesis list. The initial element of the list is the performative; the remaining elements are the performative's arguments as keyword/value pairs. Because the language is relatively simple, the actual syntax is not significant and can be changed if necessary in the future. The syntax reveals the roots of the initial implementations, which were done in Common Lisp; it has turned out to be quite flexible.

A KQML message from agent *Joe* representing a query about the price of a share of IBM stock might be encoded as:

```
(ask-one :sender Joe
         :content (PRICE IBM ?price)
         :receiver stock-server
         :reply-with ibm-stock
         :language LPROLOG
         :ontology NYSE-TICKS)
```

In this message, the KQML performative is *ask-one*, the content is *(price ibm ?price)*, the ontology assumed by the query is identified by the token *nyse-ticks*, the receiver of the message is to be a server identified as *stock-server* and the query is written in a language called *LPROLOG*. The value of the `:content` keyword is the content level, the values of the `:reply-with`, `:sender`, `:receiver` keywords form the communication layer and the performative name, with the `:language` and `:ontology` form the message layer. In due time, *stock-server* might send to *Joe* the following KQML message:

```
(tell :sender stock-server
      :content (PRICE IBM 14)
      :receiver Joe
      :in-reply-to ibm-stock
      :language LPROLOG
      :ontology NYSE-TICKS)
```

## 1.2 Functional Requirements

We arrived upon the following requirements for a KQML security model based on the analysis of the security models for Privacy Enhanced Mail [4], CORBA [3] and DCE [5]. Interested readers are referred to [2], for a thorough treatment of security threats and mechanisms to counter them. The security capabilities that should be supported include:

- *Authentication of principals.* Agents should be capable of proving their identities to other agents and verifying the identity of other agents.
- *Preservation of message integrity.* Agents should be able to detect intensional or accidental corruption of messages.
- *Protection of privacy.* The security architecture should provide facilities for agents to exchange confidential data.
- *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 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 a message.
- *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.

We also consider the following constraints or desiderata for the architecture:

- *Independence of KQML and application semantics.* The security architecture should not depend on the semantics of KQML performative (i.e., an *ask-all* from an agent will entail a *tell* or *sorry* from the receiver). The security model should be general and flexible enough to support different models of agent interaction (e.g contract net, electronic commerce).
- *Independence of transport layer.* The security architecture should not depend on the features offered by the transport layer. This is critical to facilitate agents to communicate across heterogeneous transport mechanisms and to extend the security model to accommodate embedded KQML messages.
- *No global clock or clock synchronization.* The security architecture should not be clock dependent, as global synchronization of time is difficult to achieve and would lead to further security issues of its own. Further such a security model may have inherent security weaknesses [7].
- *Authentication by crypto-unaware agents.* An agent need not have cryptographic capabilities to authenticate the sender of a message.

- *Support for a wide variety of crypto systems.* Agents should be able to use different cryptographic algorithms. But for two agents to interact, they should have a common denominator. The security architecture should not depend on any specific cryptographic algorithm.

# 2 Architecture Overview

The proposed security architecture is based on data encryption techniques [9]. In tune with the asynchronous nature of KQML, the model expects a secure message to be self authenticating and does not support any challenge/response mechanism to authenticate a message after it has been delivered. The architecture supports two security models, basic and enhanced. The basic security model supports authentication of sender, message integrity and privacy of data. The enhanced security model additionally supports non-repudiation of origin (proof of sending) and protection from message replay attacks. The enhanced security model also supports frequent change of encryption keys to protect from cipher attacks.

## 2.1 Cryptographic background

The following paragraphs define the cryptographic techniques used by this architecture and the new performative and the parameters that have been introduced to implement the architecture.

**Encryption Keys.** An agent that implements the proposed security architecture should have a master key, $K_a$, which it would use to communicate with other agents. This key can be based on a symmetric or asymmetric[1] key cryptosystem. If a symmetric key mechanism is used, we suggest that the agent, in addition to the general master key, also use a specific master key, $K_{a1,a2}$ for each agent that it communicates with, for better privacy and stronger authentication. If more than two agents share a single master key, any of those agents can masquerade as the other or eavesdrop on all the conversations between the agents sharing the key. If a master key is shared by more than two agents, the strength of security is directly related to the degree of trust between the agents.

If an agent does not share a master key, $K_{a1,a2}$ with another agent, it can use its master key, $K_a$, or can use the services of a central authentication server to generate such a key. The agents may use different keys in either direction of message flow i.e., $K_{a1,a2}$ is created by *a1* and would be used when *a1* is sending a message to *a2* and $K_{a2,a1}$ is created by *a2* and would be used when *a2* is sending a message to *a1*.

If an asymmetric key mechanism is used, a unique key for each pair of agents is not necessary, as the agent can use the public key of its peer agent to encrypt the message and prevent eavesdropping. It can also use its private key to sign the message and prove its identity to its peer. We assume that the agents know the master key of the other agents. We also suggest a secure mechanism to do master key lookup.

**Session key.** In the enhanced model, the agents use an additional key, the session key, to ensure privacy, message integrity and proof of identity. The session key can be symmetric or asymmetric and can be generated with the help of the authentication server. The session keys are set up by using a protocol explained later which requires the use of a master key to ensure security.

The agents can use either the session or master key for exchanging messages and must inform the other agent of the key that was used for encryption to ensure proper decryption.

[1] Public key cryptosystems are a very familiar example of an asymmetric key system.

When agents exchange keys, they encrypt them using the current session or master key. Keys are never exchanged in clear text form. We recommend the use of the enhanced security model [2] with an expensive master key and a cheap session key which is changed frequently.

**Message Id.** The message ID is used in the enhanced security model to protect agents from attacks by message replay. When the two agents establish a session key, they also exchange a message ID which the sender would use in the next message. Every message from an agent would carry a message ID and a new message ID for the next message. Each message ID is used only once to prevent replay and they are encrypted using the session or master key for security.

**Message Digest.** Each secure message generated using this architecture has a message digest or signature associated with it. The digest is calculated using a secure hash function like MD2, MD5 or SHS [9]. This hash function computes a digital fingerprint of the message (i.e., acts as a "checksum" for the message). The sender then encrypts this digest using the session or master key and attaches it to the message.

This encrypted message digest forms the core of the security architecture. The receiver of a message uses this digest to verify the identity of the sender and the integrity of the message. The digest also protects the message ID field from being hijacked and used in a different message.

## 2.2 Changes to KQML

In order to implement this security architecture we propose several new KQML performatives, several new parameters and some modifications to a proposed standard ontology for agents.

**Ontological assumptions.** We assume that KQML-speaking agents use a basic agent ontology which provides a small set of classes, attributes and relations helpful in talking about agents, their properties and the relationships and events in which they partake. Assuming this ontology, this architecture introduces a new sub-class of agent named *authenticator* and a new relation, *key/5* which describes a key used by an agent:

```
(key <sending-agent>
     <receiving-agent>
     <master-key?>
     <key-type>
     <encrypted-key>)
```

An instance of this relation specifies a key that the sending agent will use in secure communication with the receiving agent. If the third argument is *true* then the key is a master key, else it is a session key. If the receiving agent is a variable (e.g., ?), then the key is used by the sending agents for communication with all agents. Note that this would typically be the case for asymmetric keys.

We assume that agent addresses are represented in this ontology with the *address/3* relation:

```
(address <agent>
         <transport>
         <transport-address>)
```

Instances of this relation specify transport addresses for the agent given in the first argument, as in:

[2] This may not always be possible. The enhanced security model cannot be used if the sender of a message does not know who the intended recipient is; i.e in the case of facilitation performatives the facilitator determines the intended recipient and not the message originator.

```
(address r2d2 smtp r2d2@umbc.edu)
(address r2d2 tcpip (cujo.cs.umbc.edu 8088))
```

These addresses are known to special agents, such as *agent name servers* and *authenticator agents.*

**New parameters.** Several new KQML parameters are required to implement the security architecture.

**:auth-digest (<digest-type> <encrypted-digest>).** The *digest-type* specifies the hashing function used (MD4, MD5, etc.) to compute the message digest. The *encrypted-digest* is the message digest encrypted using the key specified by the *:auth-key* parameter. This parameter should be present to prevent message hijack, and to provide for sender authentication and integrity assurance.

**:auth-msg-id (<msg-id> <encrypted-msg-id>** This parameter is required only in the enhanced security model where it is used prevent message replay. The value is a list whose first element is the agreed upon random string, or $NIL$ if this is the first message. The second element specifies the message ID for the next message and is encrypted using the key specified by the *:auth-key* parameter. For effective prevention of message replay, this parameter should be present in each message.

**:auth-key (<bool> <key-type> <encrypted-key>).** This parameter specifies the key being used to encrypt any *:auth-digest* and *:auth-msg* parameters present. If the first element of the triple is is *true* then the master key is used[3], otherwise, the session key is used.

**New KQML performatives.** The following new KQML performatives have been added to implement the security architecture.

**auth-link.** The sender wishes to authenticate itself to the receiver and set up a session key and message ID. [4]

**auth-challenge.** The sender challenges the identity of the receiver in response to an *auth-link.* The sender encrypts a random string using the master key $K_{s,r}$ or $K_s$ and sends it as *:content.*[5]

**auth-private.** The sender is sending a confidential message to the receiver. The *:content* parameter contains the encrypted message and the *:auth-key* parameter specifies the encryption key. The *:auth-digest* parameter should be present to verify the identity of the sender and the *:auth-msg-id* and *:auth-key* parameters may be present if enhanced security model is used.

[3] An agent would use the master key for encryption if it does not share a session key with the receiving agent or if it does not know the receiver in advance. Under these circumstances, it could use this parameter to help the receiver in choosing the proper decryption key.

[4] We could eliminate the need for this performative if we are willing to send an *achieve* with an embedded *auth-challenge* but this is simply a mater of protocol detail design.

[5] Again, if we were motivated to decrease the number of performatives at the expense of putting more in the authentication ontology, we could represent an *auth-challenge* performative as an *ask-one* with *:content* of *(encrypt ?s ES)* where ES is the encrypted string and the encrypt/2 relation holds between the string with respect to the current key.

**help.** We introduce a new generic performative by which an agent can ask another for help in processing the the embedded performative given as the value of the *:content* parameter. The nature of the "help" is determined by the embedded performative and the value of the *:ontology* parameter.

If the *:ontology* is *authentication*, then a crypto-unaware agent is enlisting the help of a trusted friend to process a performative it has received, which is included as the value of the :content parameter. This embedded message can be either an *auth-link* or a generic message to be authenticated. In the case of a auth-link (i.e., a challenge), the appropriate response is a reply with a random challenge string. In the case of a message to be authenticated, the response will be an error or a reply to forward.

# 3 Basic security model

An implementation should support the following protocol to conform with the basic security model. This model supports authentication, integrity and privacy of data. If asymmetric keys are used for session and master keys, this model also supports non-repudiation of origin.

When R2D2 sends a secure message to C3PO, it would compute a message digest and encrypt it using the master key (as indicated by the value *T* for the *:auth-key* parameter).

```
<performative>
  :sender R2D2
  :receiver C3PO
  :auth-key T
  :auth-digest (<digest-type><encrypted-digest>)
  ...
```

Alternatively, if R2D2 needs to send a confidential message to C3PO, it can encrypt the message and embed it in an *auth-private* performative.

```
auth-private
  :sender R2D2
  :receiver C3PO
  :auth-key T
  :auth-digest (<digest-type> <encrypted-digest>)
  :content <encrypted-KQML-message>
```

This model can be used when R2D2 does not know the recipient in advance, e.g., for messages to be broadcast or routed by a facilitator agent, or if R2D2 and C3PO do not require prevention of message replay and can afford the cost of using the master key.

In the above message, the *:auth-digest* parameter can be used to verify the integrity of the message, authenticate the sender and ensure non-repudiation of origin (if the master key is asymmetric in nature). If the message has been corrupted, the message digest will not agree with the value of the *:auth-digest* parameter. Since the message digest is encrypted with the master key of the *:sender*, only the *:sender* or the agents with which the *:sender* shares the encryption key could have generated the message. If the master key is an asymmetric key, only the *:sender* could have generated the message, as only the *:sender* knows the private key that has been used for encryption. Note that we can only verify the identity of the generator (i.e., the message was encrypted by the *:sender* agent) of the message. This message can be a replay of a legitimate message previously sent by the generator.

# 4 Enhanced security model

The enhanced security model adds prevention of message replay, and stronger non-repudiation of message origin (if asymmetric keys are used). Even though non-repudiation can be achieved in the basic security model, we can only be sure that the message was generated by the sender, as a rogue agent can replay a message and we will not be able to detect it.

In the remainder of this section we will demonstrate how the new KQML performatives and parameters can be used to converse/communicate securely.

## 4.1 Self authentication

Agent R2D2 has cryptographic capabilities and would like to prove its identity to agent C3PO. The agents would follow the following handshake protocol to achieve it.

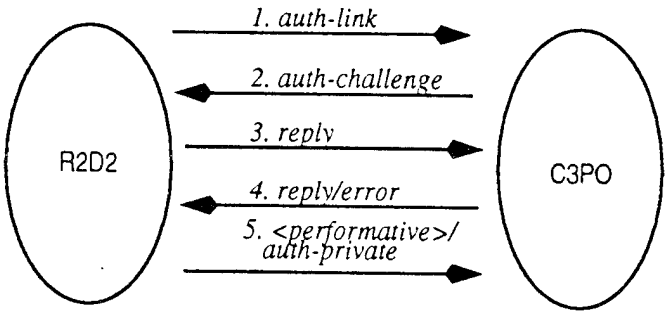



Figure 2: **The self authentication protocol is initiated by an agent who wants to establish its identity before beginning a dialog.**

R2D2 sends an *auth-link* performative to C3PO.

```
auth-link                    (1)
  :sender R2D2
  :receiver C3PO
  :reply-with <expression>
```

If C3PO will not authenticate senders, it can respond with an *error*, otherwise it sends a *auth-challenge* with a random string encrypted using the master key. A random string is used to prevent message replay.

```
auth-challenge               (2)
  :sender C3PO
  :receiver R2D2
  :in-reply-to <expression>
  :reply-with <expression>
  :content <encrypted-random-string>
```

R2D2 responds with a *reply* performative with the *:auth-digest*, *:auth-msg-id* and new session key (if present) encrypted in the master key. The value of *:content* and *:auth-msg-id* is the decrypted random string. The session key parameter is optional.

```
reply                        (3)
  :sender R2D2
  :receiver C3PO
  :in-reply-to <expression>
  :reply-with <expression>
  :auth-digest (<digest-type> <encrypted-digest>)
```

```
:auth-msg-id (<msg-id> <encrypted-msg-id>)
:auth-key (T <key-type> <encrypted-key>)
:content <random-string>
```

Now, C3PO can verify if the sender is R2D2 by inspecting the random string. Only R2D2 (or in the case of symmetric key, one of the other agents which shares the same key) could have decrypted the random string as it was encrypted using the master key. The message digest can be used for non-repudiation if asymmetric keys are used.

C3PO responds with a *reply* or an *error* depending on the success of authentication (3).

Now, R2D2 can send an authenticated message to C3PO by using the session key or master key to encrypt the message digest and a non replayable message by using the *:auth-msg-id* parameters.

```
<performative>          (4a)
  :sender R2D2
  :receiver C3PO
  :auth-digest (<digest-type> <encrypted-digest>)
  :auth-msg-id (<msg-id> <encrypted-msg-id>)
  :auth-key (<bool> <key-type> <encrypted-key>)
  ...
```

Or if R2D2 needs to send a confidential message to C3PO, it can encrypt the message and embed it in an *auth-private* performative.

```
auth-private            (4b)
  :sender R2D2
  :receiver C3PO
  :auth-digest (<digest-type> <encrypted-digest>)
  :auth-msg-id (<msg-id> <encrypted-msg-id>)
  :auth-key (<bool> <key-type> <encrypted-key>)
  :content <encrypted-KQML-message>
```

## 4.2 Authentication by request

R2D2 may expect some of the incoming messages from C3PO to be authenticated and it can initiate the authentication process by following the handshake protocol shown in figure 3. R2D2 initiates the authentication process by sending an *achieve* of an *auth-link* to C3PO.

```
achieve           (1)
  :sender R2D2
  :receiver C3PO
  :reply-with <expression>
  : content (auth-link :sender C3PO :receiver R2D2)
```

C3PO and R2D2 would then follow the steps outlined in *Self Authentication.*

## 4.3 Crypto un-aware agents

Agent Leia may not have crypto capabilities. But it trusts its friend R2D2 and R2D2 is prepared to authenticate messages on behalf of Leia. Since Leia does not have crypto capabilities, it will not accept *auth-private* performative. The agents would follow the handshake protocol given below to verify SkyWalker's identity.

Agent SkyWalker begins by sending Agent Leia an *auth-link* message to initiate the process of proving its identity to Leia.

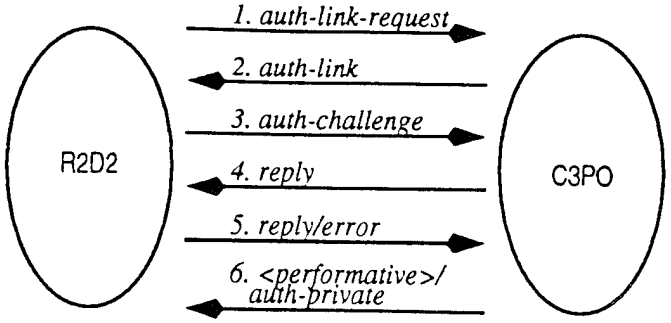



Figure 3: **R2D2 initiates the authentication by request protocol in anticipation of a dialog with agent C3PO.**

```
auth-link            (1)
  :sender SkyWalker
  :receiver Leia
  :reply-with <expression>
```

When Leia receives an *auth-link* message from SkyWalker, Leia requests a challenge string from its trusted friend, R2D2.

```
help  (2)
  :sender Leia
  :receiver R2D2
  :reply-with <expression>
  : ontology authentication
  :content (auth-link
            :sender SkyWalker
            :receiver Leia
            :reply-with <expression>)
```

R2D2 will generate a random string on behalf of Leia, encrypt it using the master key (shared by Leia and SkyWalker or Leia's master key, which R2D2 knows) and will forward it to Leia.

```
reply                (3)
  :sender R2D2
  :receiver Leia
  :in-reply-to <expression>
  :content (SkyWalker <encrypted-random-string>)
```

Leia will construct an *auth-challenge* performative and send it to SkyWalker. Subsequent performative from SkyWalker with an *:auth-digest* will be forwarded to R2D2.

```
auth-challenge       (4)
  :sender Leia
  :receiver SkyWalker
  :reply-with <expression>
  :in-reply-to <expression>
  :content <encrypted-random-string>
```

SkyWalker will respond with a secure *reply*.

```
reply                (5)
  :sender SkyWalker
  :receiver Leia
  :reply-with <expression>
```

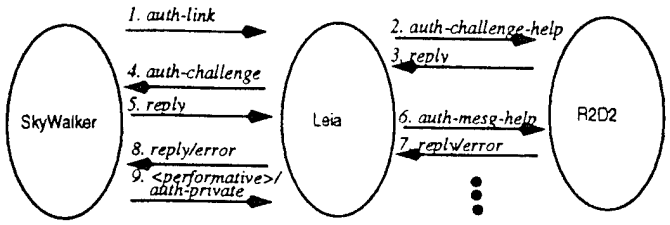



Figure 4: **Using the trusted friend protocol, agent leia asks agent R2D2 for help in authenticating agent Skywalker.**

```
:in-reply-to <expression>
:auth-digest
  (<digest-type> <encrypted-digest>)
:auth-msg-id (<msg-id>  <encrypted-msg-id>)
:auth-key (T <key-type> <encrypted-key>)
:content random-string
```

Leia will wrap the response in an *help* and send it to R2D2.

```
help          (6)
  :sender Leia
  :receiver R2D2
  :reply-with <expression>
  :ontology authentication
  :content (reply
             :sender SkyWalker
             :receiver Leia
             ... message (5) ...)
```

R2D2 will respond with a *reply* or an *error* (7). Leia would forward the R2D2's reply to SkyWalker (8). The handshake is now complete and SkyWalker can send secure performative to Leia, which Leia would verify with the help of R2D2 (9).

# 5 Authenticator Agent

The authenticator acts as a repository of the agent's master keys. It can also generate session or master keys for the agents. The security architecture does not depend on the existence of an authenticator.

An agent and the authenticator share a master key which is known only to the agent and the authenticator. The master key may actually be a pair, one for the agent to send messages to the authenticator and the other for the authenticator to send messages to the agent.

The authenticator accepts only messages in the enhanced model, i.e., the messages should have an *:auth-msg-id*. So, each agent should have established a secure link using *achieve* of *auth-link* and *auth-link* with the authenticator upon startup. It is the agent's responsibility to verify the identity of the authenticator and prove its identity to the authenticator.

## 5.1 Key lookup using the Authenticator

Agent Solo has received a message from Chewie and would like to know the master key used by Chewie. Solo uses the following protocol to get the master key from the authenticator.

Agent Solo would send an *ask-one* to the authenticator to lookup the master key used by Chewie to send out messages. The *:content* parameter contains the requested key-type.

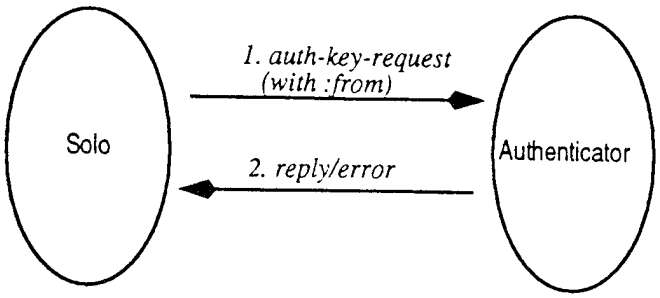



Figure 5: **Agent solo has received a message from agent Chewie and asks the authenticator for the appropriate key for decryption.**

```
ask-one          (1)
  :sender Solo
  :receiver Authenticator
  :reply-with <expression>
  :auth-digest (<digest-type> <encrypted-digest>)
  :auth-msg-id (<msg-id> <encrypted-msg-id>)
  :auth-key (<bool> <key-type> <encrypted-key>)
  :content (key solo Chewie T ?key-type ?key)
```

If Chewie had previously registered a master key for communication with Solo, the authenticator will return that key in a *tell* performative. If there is no such key, the authenticator will reply with an *sorry*.

```
tell
  :sender Authenticator
  :receiver Solo
  :in-reply-to <expression>
  :auth-digest (<digest-type> <encrypted-digest>)
  :auth-msg-id (<msg-id> <encrypted-msg-id>)
  :auth-key (<bool> <key-type> <encrypted-key>)
  :content (key solo Chewie T
                <key-type> <encrypted-key>)
```

## 5.2 Key creation using the Authenticator

Agent Solo would like to send a secure message to Chewie and needs a session or master key for that purpose. It can send an *ask-one* to the authenticator to create such a key.[6] If a master key has been requested, the authenticator would store the key in its database. A master key creation would not be necessary if asymmetric keys are used as a single master key per agent is suffice to talk securely to all the agents. Further, non-repudiation of message origin is not possible if the authenticator knows the private key.

Agent Solo would send an *ask-one* to generate a master or session key to send messages to Chewie.

```
ask-one                    (1)
  :sender Solo
  :receiver Authenticator
  :reply-with <expression>
  :to Chewie
```

[6]Note that this is not ruled out by the semantics of KQML and is done by convention by authenticator agents. If a registered agent asks an authenticator for a key which is not in the authenticator's database, it will create one and then tell the agent about it.

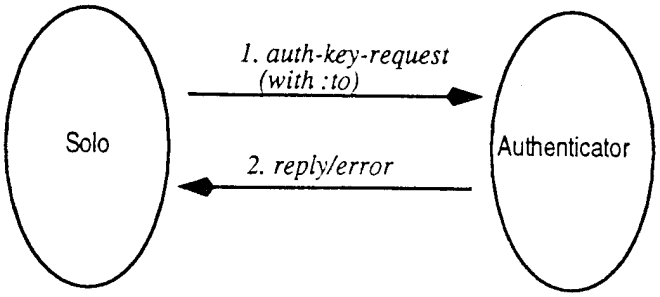



Figure 6: **Agent solo requests an appropriate key for sending agent Chewie a secure message.. The authenticator will create the key if necessary.**

```
:auth-digest (<digest-type> <encrypted-digest>)
:auth-msg-id (<msg-id> <encrypted-msg-id>)
:auth-key (<bool> <key-type> <encrypted-key>)
:content (key solo Chewie <bool> ?key-type ?key)
```

Authenticator creates a key and sends it in a *tell* performative. If the requested key is a master key, the key is added to Solo's key list. If the authenticator is not able to create the key for whatever reason, it responds with an *error* performative.

```
tell                                (2)
  :sender Authenticator
  :receiver Solo
  :in-reply-to <expression>
  :auth-digest (<digest-type> <encrypted-digest>)
  :auth-msg-id (<msg-id> <encrypted-msg-id>)
  :auth-key (<bool> <key-type> <encrypted-key>)
  :content (key solo Chewie <bool>
            <key-type> <encrypted-key>)
```

## 5.3 Key registration with Authenticator using KQML

Agent Yoda would like to register its master keys with the authenticator.

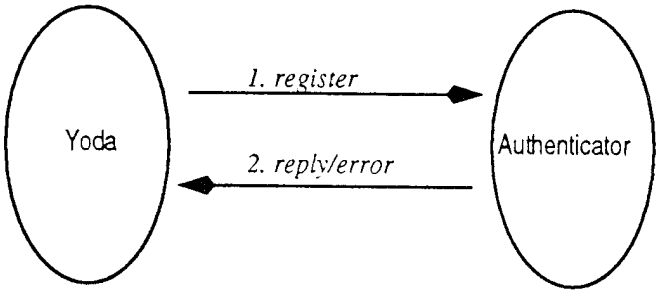



Figure 7: **Agent Yoda registers its keys with the authenticator. This presupposes that Yoda is able to establish a secure communication link.**

Yoda would send a secure register with the keys encrypted using the key specified by the *:auth-key* parameter. The agent can also use this performative to change the master key that it shares with the authenticator.

```
register                            (1)
  :sender Yoda
  :sender-address (tcpip (tcp-host tcp-port))
```

```
:receiver Authenticator
:reply-with <expression>
:auth-digest (<digest-type> <encrypted-digest>)
:auth-msg-id (<msg-id> <encrypted-msg-id>)
:auth-key (<bool> <key-type> <encrypted-key>)
:ontology authentication
:content (and (key Yoda <a1> <ktype1> <ekey1>)
              (key Yoda <a1> <ktype1> <ekey1>)
              ...))
```

Note that Yoda sends its address using the *:sender-address* parameter. If the key registration is successful, the authenticator responds with a *reply* (2a) else with an *error* (2b).

### 5.4 Initial key registration with the authenticator

Agent Yoda is starting up for the first time and would like to register the master key that it shares with the authenticator. This can be achieved either using KQML or some other external mechanism.

If symmetric keys are used, KQML cannot be used to register the initial key as there is no master key to encrypt the key. If asymmetric keys are used, the initial master key is encrypted using the authenticator's public key. Even if asymmetric keys are used, there is a security problem. A rogue agent, agent DarthVader may know that agent Ben respects performative from agent Yoda. Agent DarthVader may also find out that Yoda has not registered with the authenticator and therefore the authenticator does not know the existence of such an agent. Now, DarthVader can register itself as Yoda. If this type of masquerading can be an issue, KQML should not be used for the initial registration.

The protocol would be same as the key register process. The *:content* parameter will contain only one key relation and the agent name would be NIL as this is an asymmetric key and it is sufficient to use a single asymmetric master key for all the agents. If the authenticator does not have any entry for Yoda, it accepts the registration and adds it to its database and sends a *reply*.

## 6 Limitations of this model

The security model we are proposing for KQML has a number of limitations which we briefly enumerate. Some of the limitations stem from the basic features of KQML, others are reasonable and, we believe, can be lived with and the rest could be addressed by if required by revising the architecture.

- *Credentials.* The model does not provide a mechanism to exchange credentials – that is, for one agent to empower another to act on its behalf. Lets say that agent Emperor trusts agent DarthVader and would like to delegate DarthVader to act on its behalf. There is no way for DarthVader to take up Emperor's credentials.

- *Non-repudiation of receipt.* The model does not support non-repudiation of message receipt. This can be a very useful capability but would be difficult to implement due to the asynchronous nature of KQML and can be done only at the application level.

- *Messages to unknown receivers.* Although our enhanced model does support *message replay detection*, the proper use of the *:auth-msg-id* parameter is required. This requires that the recipient is known in advance. One of the essential features of KQML is the use

of facilitator class agents (e.g., brokers and proxy agents) to automatically rout messages whose intended recipients are described in general terms by the sending agent.

- *Introduction of state.* The security architecture requires that agents maintain state information. Agent Emperor has to keep track of the next message ID and optionally the next session key that will be used by agent DarthVader. The agents can choose not to use this feature if they are not concerned with message replay attack and cipher attack. In some software architectures (e.g., KATS) the state information can be handled by a generic sub-agent, freeing the end application agent of the burden.

- *Crypto-awareness.* An agent can send out authenticated messages if and only if it has crypto capabilities. Again, a good software architecture can build the crypto-awareness into a generic trusted sub-agent.

- *Constraints on delivery.* Messages delivery must be reliable and in order. (A fair limitation considering that KQML itself assumes that).

- *No protection from traffic analysis.* The architecture does not address traffic analysis by rogue agents. We feel that traffic analysis is best handled at the link/transport layers.

- *Use of recommended APIs.* The model should be enhanced to support the use of the Crypto APIs recommended by NSA (GSS, GCS and Cryptokit) [8], especially for the key-type and digest-type values.

# 7 Implementation

We have proposed a security model which has not yet been implemented. We have examined the modifications which would be required to integrate this model into some existing KQML API software. We will briefly discuss the approach to implementing this architecture in the KATS KQML API [10, 23] software architecture.

In this architecture, each agent application has a separate router sub-agent. The routers used by all the agents are identical and handle all KQML messages going to and from the agent. The security enhancement can be easily added to this KQML implementation by modifying the router to be security aware, without involving any major change to the agent application.

The agent application only needs to specify the degree of security (any combination of provide for message authentication, protect from replay attack, send a confidential message and sign the message-non-repudiation of origin) of an outgoing message. The router would handshake with the receiving agent and secure the message to the extent possible (the receiving agent may not support asymmetric key cryptography, *auth-private* performative etc or the router may not know the receiving agent of the embedded message if it is sending out a broadcast or facilitation performative).

Similarly, when the router receives an authenticate request from another agent, it can handle the handshake itself, without involving the agent application. When the router receives a message from another agent, it would tag the message with a security level (confidential, authenticated, etc.). The agent application can decide to process or ignore the message based on the message's security level.

A similar approach can be followed to add security enhancement to most other KQML implementations [15, 18]. Most implementations would provide a library with at least a basic send and receive primitive to send and receive KQML messages. These primitives can be modified to add the authentication information to the outgoing messages or process the authentication information in the incoming messages. The implementations can use one of NSA

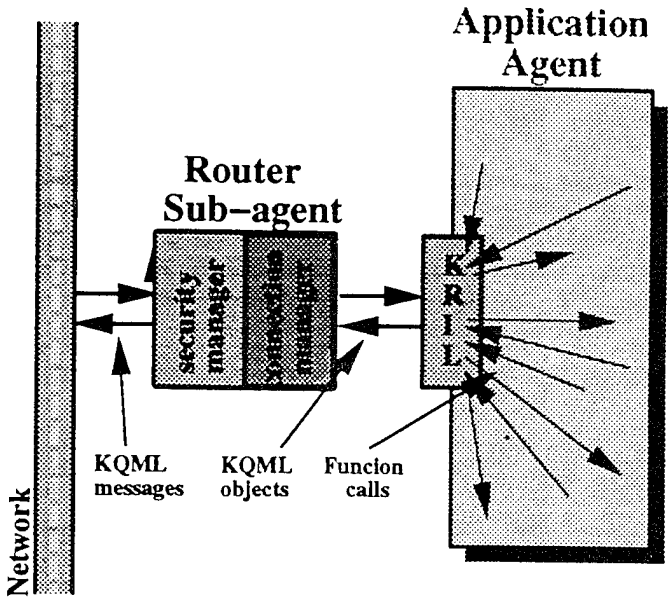



Figure 8: **The KATS system includes a router sub-agent which can handle most of the security messages automatically, freeing the agent and its authors from having to worry about security.**

recommended crypto APIs [8] for cryptographic capabilities. These APIs provide support for asymmetric and symmetric key cryptography, message digest, key generation etc. The use of a standard API would help agents using different KQML implementations to interact without any incompatibility problems.

## 8 Conclusion

The proposed security model addresses privacy, authentication and non-repudiation (if asymmetric key mechanism is used for the master and session keys) in agent communication. It does not fully address the issue of message replay, especially if the recipient of a performative is not known in advance. Ultimately, this security model depends on the strength of the crypto algorithm, message digest function and the random number generator used by the agent for its effectiveness.

## References

[1] Draft specification of the KQML agent communication language, Tim Finin, Jay Weber et al, Jun 15 1993, http://www.cs.umbc.edu/kqml/kqmlspec/spec.html

[2] Security Mechanisms in High-Level Network Protocols, Victor L.Voydock, Stephen T. Kent. ACM Computing Surveys, Vol.15, No. 2, 135-171, Jun 83

[3] OSTF RFP3 Submission, Corba Security, OMG Document Number 95-3-3, Mar 8 1995. http://www.omg.org/docs/95-3-3.ps

[4] Privacy Enhancement for Internet Electronic Mail: Part I: Message Encryption and Authentication Procedures, J. Linn, Oct 02 1993, http://ds.internic.net/rfc/rfc1421.txt

[5] Security in a Distributed Computing Environment, OSF-O-WP11-1090-3, http://www.osf.org/comm/lit/OSF-O-WP11-1090-3.ps

[6] Project Athena Technical Plan, Section E.2.1, Kerberos Authentication and Authorization System, S.P.Miller, B.C.Neuman, J.I.Schiller and J.H.Saltzer, Oct 27 1988, ftp://athena-dist.mit.edu/pub/kerberos/doc/techplan.PS

[7] Limitations of the Kerberos Authentication System, S.M. Bellovin, M. Merritt, Proceedings of the Winter 1991 Usenix Conference, Jan 1991, ftp://research.att.com/dist/internet_security/kerblimit.usenix.ps

[8] Security Service API: Cryptographic API Recommendation, NSA Cross Organization, CAPI Team, Jun 12 1995, http://www.omg.org/docs/95-6-6.ps

[9] RSA Labs' frequently asked questions (FAQ), http://www.rsa.com/rsalabs/faq

[10] Software Design Document for KQML, Revision 3.0, Mar 1995, LORAL Corporation, Paoli PA, USA

[11] Tim Finin, Don McKay, Rich Fritzson, and Robin McEntire. KQML: an information and knowledge exchange protocol. In Kazuhiro Fuchi and Toshio Yokoi, editors, *Knowledge Building and Knowledge Sharing*. Ohmsha and IOS Press, 1994.

[12] Tim Finin, Don McKay, Rich Fritzson, and Robin McEntire. The KQML information and knowledge exchange protocol. In *Third International Conference on Information and Knowledge Management*, November 1994.

[13] Michael Genesereth and Richard Fikes. Knowledge interchange format, version 3.0 reference manual. Technical report, Computer Science Department, Stanford University, June 1992.

[14] Mike Genesereth. An agent–based approach to software interoperability. Technical Report Logic–91–6, Logic Group, CSD, Stanford University, February 1993.

[15] Michael R. Genesereth and Steven P. Katchpel. Software Agents. newblock Communications of the ACM, v37, n7, pp 48–53, 147, 1994.

[16] Daniel R. Kuokka, James G. McGuire, Jay C. Weber, Jay M. Tenenbaum, Thomas R. Gruber, and Gregory R. Olsen. Shade: Technology for knowledge–based collaborative. In *AAAI Workshop on AI in Collaborative Design*, 1993.

[17] Yannis Labrou and Tim Finin. A semantics approach for KQML – a general purpose communication language for software agents. In *Third International Conference on Information and Knowledge Management*, November 1994. Available as `http://www.cs.umbc.edu/-kqml/papers/kqml-semantics.ps`.

[18] James G. McGuire, Daniel R. Kuokka, Jay C. Weber, Jay M. Tenenbaum, Thomas R. Gruber, and Gregory R. Olsen. Shade: Technology for knowledge–based collaborative engineering. *Journal of Concurrent Engineering: Applications and Research (CERA)*, 1(2), September 1993.

[19] R. Neches, R. Fikes, T. Finin, T. Gruber, R. Patil, T. Senator, and W. Swartout. Enabling technology for knowledge sharing. *AI Magazine*, 12(3):36–56, Fall 1991.

[20] Ramesh Patil, Richard Fikes, Peter Patel-Schneider, Donald McKay, Tim Finin, Thomas Gruber, and Robert Neches. The DARPA knowledge sharing effort: Progress report. In B. Nebel, C. Rich, and W. Swartout, editors, *Principles of Knowledge Representation and Reasoning: Proc. of the Third International Conference (KR'92)*, San Mateo, CA, November 1992. Morgan Kaufmann.

[21] M.Tenenbaum, J. Weber, and T. Gruber. Enterprise integration: Lessons from shade and pact. In C. Petrie, editor, *Enterprise Integration Modeling*. MIT Press, 1993.

[22] Chelliah Thirunavukkarasu. A Security Architecture for KQML. Technical Report MS-EECS-95-nn, Computer Science and Electrical Engineering Department, University of Maryland Baltimore County. August, 1995.

[23] KQML_Agent Technology Software. UMBC technical report. 1995.

# On Agent Domains, Agent Names and Proxy Agents[1]

**Tim Finin and Anupama Potluri**
**Computer Science and Electrical Engineering**
**University of Maryland Baltimore County**
**Baltimore MD**

**Chelliah Thirunavukkarasu[2]**
**Enterprise Integration Technology**
**Palo Alto, CA**

**Don McKay and Robin McEntire**
**Valley Forge Engineering Center**
**Loral Government Systems Group**
**Paoli PA**

We consider the problem of how software agents should be named and what kind of software infrastructure is necessary in order to locate an agent given only its name. We assume an agent environment which (1) is dynamic with agents being created and destroyed frequently; (2) undergoes re-organizations with agent groups and sub-groups forming and disbanding; and (3) supports agent communication by any of several transport mechanisms such as TCP/IP, email, http and distributed object systems. This leads us to propose the establishment of *agent domains* which are organized into an *agent domain hierarchy*. Agent name resolution can be done by *agent name servers*, analogous to Internet *domain name servers*. One of the additional benefits from this approach is that it easily supports the definition of *proxy agents*. We sketch how this architecture would impact the KQML agent communication language and protocol and describe an implementation of a generic *KQML Agent Name Server* and its integration into the KATS framework.

Submitted to AAAI-96 with tracking number **A603** under content areas **distributed AI, multi-agent systems, and software agents** with approximately 4000 words.

**Acknowledgements:** This work has been the result of very fruitful collaborations with a number of colleagues with whom we have worked on KQML and other aspects of the Knowledge Sharing Effort. We wish to specifically thank and acknowledge James Mayfield, Richard Fritzson, Charles Nicholas, and R. Scott Cost.

[1] This work was supported in part by the DoD, the Air Force Office of Scientific Research under contract F49620-92-J-0174, and by the Advanced Research Projects Agency monitored under USAF contracts F30602-93-C-0177 and F30602-93-C-0028 by Rome Laboratory.

[2] This work was done while the author was at UMBC.

# On Agent Domains, Agent Names and Proxy Agents[3]

We consider the problem of how software agents should be named and what kind of software infrastructure is necessary in order to locate an agent given only its name. We assume an agent environment which (1) is dynamic with agents being created and destroyed frequently; (2) undergoes re-organizations with agent groups and sub-groups forming and disbanding; and (3) supports agent communication by any of several transport mechanisms such as TCP/IP, email, http and distributed object systems. This leads us to propose the establishment of *agent domains* which are organized into an *agent domain hierarchy*. Agent name resolution can be done by *agent name servers*, analogous to Internet *domain name servers*. One of the additional benefits from this approach is that it easily supports the definition of *proxy agents*. We sketch how this architecture would impact the KQML agent communication language and protocol and describe an implementation of a generic *KQML Agent Name Server* and its integration into the KATS framework.

Submitted to AAAI-96 with tracking number **A603** under content areas **distributed AI, multi-agent systems, and software agents** with approximately 4000 words.

## Introduction

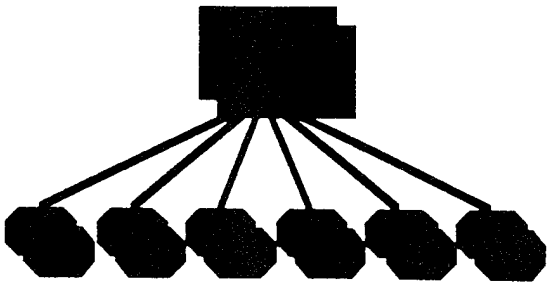
Agents need to talk to other agents. If you are an agent A and there is another specific agent B that you want to send a message to, how do you manage it? Clearly there is a need for some kind of referential expression that A can use for B and which can be given to the underlying machinery which will convey the message to B. One solution is to use an expression that locates the agent with respect to the message transport system. Examples of such "transport address names" would be a structure which contains an IP address and a port number, or a URL, or an email address for the TCP/IP, http and SMTP protocols. This is a common practice in many of our primitive agent systems today.

Another approach allows agents to use one of more symbolic names and to provide some kind of mechanism by which names can be registered and associated with their appropriate "transport address name". This approach is only slightly more sophisticated than the first. The name registration can be done in any of several ways, such as hand coding the associations into all of the agents, or broadcasting the associations over the transport mechanism or assuming the use of "communication facilitator"[4] type agents.

---

[3] This work was supported in part by the DoD, the Air Force Office of Scientific Research under contract F49620-92-J-0174, and by the Advanced Research Projects Agency monitored under USAF contracts F30602-93-C-0177 and F30602-93-C-0028 by Rome Laboratory.

[4] Genesereth (Genesereth 95) introduced the term *"communication facilitator"* or *"facilitator"* to refer to agents which perform various message-oriented services for other agents such as content-based routing.



agents registered with agent name server ANS1

**Figure 1 -- a set of agents are associated with an agent name server by sending it a KQML "register" performative.**

Agent domains will be organized into a hierarchy. Agents will register with an ANS, as shown in figure one. An ANS, being an agent itself, will register with a "parent ANS", resulting in a hierarchy, as shown in Figure Two. Each agent will have one or more local names. An agent can also be referred to by its "domain qualified name". For example, consider the agent-domain hierarchy in Figure Three.

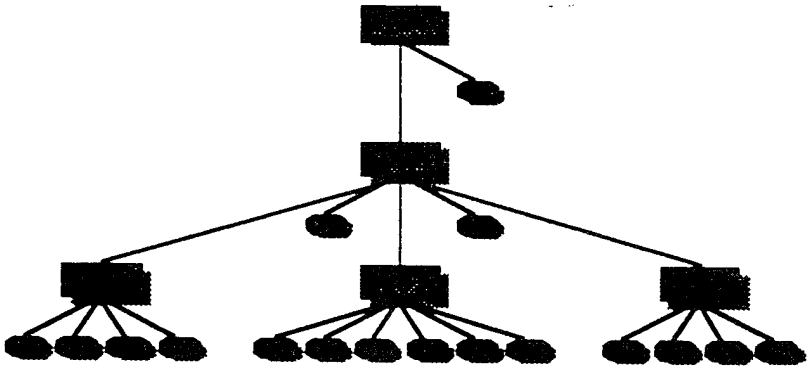


**Figure 2 -- Agent name servers are organized into a hierarchy through the registration process.**

One possibility we might consider is just to "piggy-back" on the existing Internet host structure. For example, why not refer to the agent "colossus" running on the machine "cujo.cs.umbc.edu" as "colossus@cujo.cs.umbc.edu" and assume a standard port for KQML speaking agents. This idea is attractive in that it makes efficient use of a well thought out and implemented architecture. However, there are several problems which argue against this. The primary difficulty is that we do not want to tie KQML and agent communication in general to a single transport mechanism. Current research groups are using a variety of mechanisms to carry KQML messages -- TCP/IP, SMTP, CORBA objects, and HTTP. We would like to continue to keep KQML flexible in this regard. A consequence of this is that we need a general mechanism for naming agents that is independent of the transport mechanism.

## What should agent names look like?

The KQML language and protocol includes special commands (the register and unregister performatives) by which agents can announce the symbolic names by which they wish to be known. Special agents (commonly known as "communication facilitators") traffic in this knowledge and provide a name registration and resolution service. In the Loral/UMBC "KQML Agent Technology Software" (KATS) architecture, this name registration and resolution is handled automatically by a generic router sub-agent attached to each agent. From the agents perspective, all it has to do is to specify the set of symbolic names it wished to be known by. The router sub-agent automatically contacts the local "facilitator agent" [5] to register the agent by its symbolic names.

For example, suppose the agent named A wants to send a query to agent B. It passes a KQML form like

(ask-one :from A :to B :content ...)

to its router sub-agent (call it r(A)). This router is responsible, among other things, for resolving the agent name B into an address that can be given to the transport layer for delivery. In KATS, the router checks it's cache to see if it knows how to deliver a message to an agent named "B". If it does, it ships the message out. If not, it sends a KQML query to the local agent name server, asking for the address of an agent named "B". Upon receiving the information, it adds it to its cache and sends off the message.

There are additional wrinkles, of course, such as how to determine when a cache entry is stale and needs to be flushed, but this describes the current arrangement.

## The Problem

Although this approach works quite well as far as it goes, it just does not go very far. The problem is that it only supports communication between two agents if they both register with a common agent name server. There are several possible solutions. All agents could use a single master name server possibly located deep under Cheyenne Mountain. Another approach is to have the name servers share their registration databases. Still a third, and more general, technique involves having the name servers use a distributed protocol to seek out the contact information on non-local agents. We next describe our protocol for such a distributed agent name resolution scheme.

## Distributed agent-name resolution

We propose to organize agents into "agent domains" in much the same way that the Internet is organized into "host domains". An "agent domain" can be thought of as a collection of agents that are associated with a particular set of facilitator-class agents. In particular, every agent domain must have an "agent domain name server" (or "agent name server" or ANS for short) running. There may be other facilitator-class agents, such as brokers, associated with the agent-domain.

---

brokering, forwarding, translation, etc. Facilitators use a knowledge-base of information about the information requirements and serviced of other agents together with domain knowledge to do their job.

[5] How does it find it? Well, our current implementation assumes an environment variable which points to it. An alternative convention are possible, such as assuming that a fcilitator can be reached via a standard name and address with respect to a particular transport mechanism (e.g., facilitator@domain for SMTP).

We propose a naming scheme similar to the one used for hosts on the Internet. Every agent will have one or more local names optionally followed by a domain qualifier. A local name can be any non-zero length sequence of characters chosen from a fixed character set[6]. A domain qualifier begins with the character "." and consists of one or more agent domain names separated with a "." character. Thus a fully qualified agent name has the structure:

<local name> . <domain1>.<domain2>.<domain3>...<domainN>.

The following would all be valid names for an agent with the local name "colossus" registered in the "umbc.edu." agent domain (and assuming that it is in turn registered in the "edu." domain which is in the top-level "." domain.)[7]

colossus
colossus.umbc
colossus.umbc.edu.

Furthermore, we propose a correspondence between the names of agent domains and agent domain servers. Thus in the above example, agent *colossus* is registered with the ANS with local name *umbc* which is registered with the ANS with local name *edu* which is registered with the global ANS. Thus, the fully qualified name of an agent could be defined by its local name followed by a "." followed by the fully qualified name of its official agent name server.

There are obvious alternatives to the syntax we are proposing which would model agent names after email addresses (e.g., colossus@umbc.edu) or URLs (e.g., kqml://umbc.edu/colossus). There are several arguments against using either of these existing formats. One argument that applies to both is that we would like to avoid confusion about what a particular address means, e.g., is it the name of an agent, a reference to a document, or a reference to a mailbox. One might think that such a confusion could be a feature rather than a bug, since each of these might be a very reasonable way to think of and interact with an agent. However, there is clarity to be gained by separating the concept of a abstract reference to an agent that is independent of communication channel and a reference to an agent that implies a means of communication. The email style address has an advantage of using a special character (the @) to separate the "local name" from the "host name". When standards for SMTP were being developed, this was quite useful since it provided a mechanism to support gateways between email systems that used very different protocols[8].

## How agent names are resolved

The process of resolving a name is similar to the one used for the Internet DNS. One difference is that agent with a given name can have many addresses -- one for each transport mechanism that it can use. Thus, the agent_address is a function from agent names and agent transport types to transport addresses. We assume that an agent can be referred to using its fully qualified name[9] or any non-ambiguous abbreviation.

---

[6] Say the character set {a-z,A-Z,0-1,-,_,.,+,#}.

[7] An alternate is possible, such as one based on URLs. For example, we might choose to represent an agent using "*agent://<domain>.<domain>...<domain>/<local name>*" as in the example "agent://umbc.edu.kqml/colossus".

[8] In the 1980's it was very common to see complicated email addresses that might involve several intervening gateways, such as foo@quux%bar@umbc.edu. For the most part, the necessity for these has gone away.

[9] i.e., its local name followed by a dot followed by the fully qualified name of its agent name server.

Suppose agent $A_1$ wants to resolve the fully qualified name $N_2$ into an address for transport type $T_2$. The process starts when $A_1$ asks its agent name server.[10] The query is passed up the hierarchy of agent name servers as long as the address is not known and $N_2$, is not recognized as being the name of some descendant. If an agent name server gets the query and knows the address, the process stops and a response is sent to $A_1$. If the root of the agent domain hierarchy is reached and the address was not found, the process fails and an appropriate error message is sent to $A_1$. If an agent name server recognizes that $N_2$ is the name of some descendant, it is passed down to the appropriate immediate child agent name server. This process continues until we find an agent name server that knows the address or we recognize that we can go no further. In this latter case, the process fails and an appropriate error message is sent to $A_1$.

Resolving partially qualified agent names follows a very similar process. There are a number of details that must be decided on in standardizing this name resolution protocol -- i.e., whether answers are sent directly back to the agent initiating the query or passed back through the hierarchy and cached along the way. These details should only effect the performance of the name resolution process.

## Taking names seriously

Agents should take names seriously. What we mean by this is that application agents should always refer to other agents by their names, and not the underlying transport addresses, if known.[11] Agents should leave the resolution of these names into transport addresses up to specialized agents (e.g., agent name servers) and sub-agents (e.g., routers). Adapting this convention will directly support the concept of a *proxy agent* , the use of logical agent services, and other important notions. We will discuss the concept of a proxy agent in more detail and sketch how it can be easily implemented by adopting a few simple conventions for agent name servers.

## Proxy agents and their protocols

A *proxy agent* is an agent that handles all of the incoming and outgoing messages (perhaps with respect to a particular transport mechanism) for another agent. A simple proxy mechanism can be used to provide a number of services:

- *firewall gateways* -- agents which are behind a security firewall and use a proxy agent to communication to agents outside the firewall.
- *protocol gateways* -- An agent which is unable to send or receive messages via a particular transport mechanism (e.g., email) can still communicate with agents who only use that mechanism by having a proxy agent to mediate between two transport mechanisms.
- *message processing* -- The proxy can provide a processing service, such as logging incoming or outgoing messages, without altering the stream.

---

[10] How does an agent know the address of its agent name server? Since the process has to ground out somewhere, we assume that an agent knows at least the address of its agent name server upon creation or can access it via some standard environment variable, as is done in the KATS framework.

[11] Of course, some agents, such as agent name servers, will know transport addresses and some agent components (such as a router sub-agent) will have to know and use transport addresses. Moreover, we would expect a such sub-agents to remember the transport address for a given agent name and to use it for an extended session.

- *filtering and annotating* -- The proxy can alter the stream by filtering out certain incoming messages, blocking outgoing messages to particular destinations, annotating incoming messages, etc.
- *agent composition* -- A proxy agent facility allows one to develop a notion of "agent composition" similar to functional composition in which agents take in a stream of messages and emit a possibly modified stream.
- *groupware for agents* -- Since multiple agents can share a proxy and proxies can have procy agents, it is possible to develop a hierarchy of proxy agents tto provide group oriented services for client agents, such as mediation, translation and annotation.

As an example, suppose we have two agents A and B, both of which use the agent name server F. A has proxy agent p(A) and B has proxy agent p(B). Suppose A wants to send a message to B. The following events take place:

1. A hands off the message to its router subagent r(A).
2. r(A) asks F for B's address.
3. F gives r(A) the address of p(A), A's proxy.
4. r(A) delivers the message to p(A) but the :TO field equals B.
5. p(A), knowing that it is a proxy for a (possibly among others) and noticing that it has received a message from a with the :TO field of B, understands that the message is not really intended for it, and asks its router r(p(A)) to deliver it to B.
6. r(p(A)) asks F for the B's address.
7. F gives r(p(A)) the address of p(B) -- B's proxy.
8. r(p(A)) delivers the message to p(B) with the :TO field equals B.
9. p(B), knowing that it is a proxy for B (possibly among others) and noticing that the :TO field is B, understands that the message is not really intended for it, and asks its router r(p(B)) to deliver it to B.
10. r(p(B)) asks F for the address of B.
11. F recognizes that p(B) is B's proxy so it gives p(B) the real address of B.
12. r(p(B)) caches the address and delivers the message to B.

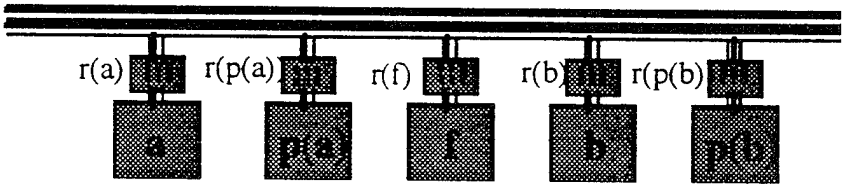



**Figure 3 -- A conversation among five agents and their sub-agents.**

This example demonstrates the use of proxy agents for both outgoing and incoming messages. The proxy agents may do some additional processing of the messages they get, of course, like logging or traffic analysis, etc. The scenario above is the worst case in that it assumes all of the router subagent caches are empty. Subsequent communications would find the caches filled, so the agent name server would not have to be involved[12].

Implementing the concept of a proxy agent is rather trivial once we have agent name servers and agents who contact agents by name rather than by transport address. First, if an agent

[12] Delivering a subsequent messge from A to B would, for example, only involve steps 1, 4 , 8 and 12.

P is willing to serve as a proxy agent, it has to be able to provide some of the functionality that an agent name server does. Second, if A wishes to use P as a proxy for transport mechanism T, it must (1) get permission from ask P for this and (2) unregister with A's agent name server for transport T (if it was so registered). Third, P should register with A's agent name server *in A's name* for transport T. Good design dictates that all of the agents involved should also explicitly "know" that P is acting as A's proxy with respect to messages carried by transport mechanism T.[13]

## Changes to KQML and standard utility agents

This naming scheme will not require any major changes to KQML such as the addition of new performatives or new parameters. It will have an impact on the form of the register performative and on the standard agent ontology and on the protocols used by standard utility agents such as an agent name server and a router. This, in turn, will effect the protocols that all agents who use these standard utility agents follow.

An agent name server will have to store more information about the agents that are registered with it and will have to handle some additional performatives. When an agent registers with an agent name server, it should provide a set of symbolic names it will respond to and a set of transport type/address pairs. Authentication information may be provided as described in (Thirunavukkarasu, Finin and Mayfield 95). A standard agent name server must handle requests to register and unregister from agents as well as various kinds of queries against its registration database.[14]

In reaching a consensus on the precise details of how to add these changes to KQML we will have to choose what aspects are expressed by adding to or modifying the basic components of KQML (i.e., performatives and parameters and their semantics) and which are expressed by extending the common "*agent ontology*" that is assumed by KQML.[15]

## Conclusions

We have discussed the problem developing a global naming scheme for software agents and how such names can be resolved into usable addresses. We have assumed an agent environment which (1) is dynamic with agents being created and destroyed frequently; (2) undergoes re-organizations with agent groups and sub-groups forming and disbanding; and (3) supports agent communication by any of several transport mechanisms such as TCP/IP, email, http and distributed object systems. We proposed the use of *agent domains* which are organized into an *agent domain hierarchy*. Agent name resolution will be done *agent name server* agents which use a distributed protocol similar to that used by Internet *domain name servers*. This approach supports the definition of *proxy agents* which have a variety of uses. We have briefly discussed how this proposal would impact the KQML agent communication language and protocol and describe an ongoing implementation of a generic *KQML Agent Name Server* and its integration into the KATS framework.

---

[13] This will allow the agents to recognize, for example, that messages delivered to P's address but addressed to A were not sent, delivered or addressed in error.

[14] A careful specification of the requirements for an agent name server would include the performatives register, unregister, ask-one, ask-all, stream, subscribe and deny.

[15] This ontology is only now being articulated by defining it using the Ontolingua language.

## Bibliography


Paul Albitz & Cricket Liu, *DNS and BIND*, O'Reilly, 1992, ISBN:1-56592-010-4.

Keith Clark and Frank McCabe, *April -- Agent Process Interaction Languge,* CIKM Workshop on Intelligent Information Agents, Baltimore, 1995.

Tim Finin, Don McKay, Rich Fritzson, and Robin McEntire. The KQML information and knowledge exchange protocol. In Third International Conference on Information and Knowledge Management, November 1994.

Tim Finin, Yannis Labrou, and James Mayfield, KQML as an agent communication language, invited chapter in Jeff Bradshaw (Ed.), ``Software Agents'', MIT Press, Cambridge, to appear, (1995).

Rich Fritzson, Tim Finin, Don McKay and Robin McEntire. KQML - A Language and Protocol for Knowledge and Information Exchange, 13th International Distributed Artificial Intelligence Workshop, July 28-30, 1994. Seattle WA.

Michael R. Genesereth and Steven P. Katchpel, *Software Agents,* CACM, v37 n7, pp 48--53, 147, 1994.

Mark R. Horton, *What is a domain?,* available on-line as <http://www.dns.net/dnsrd/docs/domain.ps>.

Yannis Labrou and Tim Finin. A semantics approach for KQML—a general purpose communication language for software agents. In Third International Conference on Information and Knowledge Management, November 1994. Available on-line as http://www.cs.umbc.edu/kqml/papers/kqml-semantics.ps.

R. Patil, R. Fikes, P. Patel-Schneider, D. McKay, T. Finin, T. Gruber, and R. Neches. The DARPA knowledge sharing effort: Progress report. In B. Nebel, C. Rich, and W. Swartout, editors, Principles of Knowledge Representation and Reasoning: Proc.\ of the Third International Conference (KR'92), San Mateo, CA, November 1992. Morgan Kaufmann.

R. Neches, R. Fikes, T. Finin, T. Gruber, R. Patil, T. Senator, and W. Swartout. Enabling technology for knowledge sharing. AI Magazine, 12(3):36--56, Fall 1991.

James Mayfield, Yannis Labrou and Tim Finin. Evaluation of KQML as an Agent Communication Language, the IJCAI-95 Workshop on Agent Theories, Architectures, and Languages.

M. Wooldridge and J. P. Muller and M. Tambe (eds.), *Intelligent Agents Volume II --- Proceedings of the 1995 Workshop on Agent Theories, Architectures, and Languages (ATAL-95)* , Springer Verlag, 1996.