

REQUIREMENT SPECIFICATION FOR THE DECAF-BROKER

Mikko Laukkanen, Jukka Eskelinen
University of Delaware
Department of Information and Computer Science

May 1, 1999

Document history

Version	Editor	Date	Comments
0.1 (DRAFT)	Jukka Eskelinen	Wed Apr 7 13:57:05 EDT 1999	The first version

1 Introduction

This document specifies the requirements for the DECAF-Broker, which will be included in the DECAF Agent Framework. Next sections are organized as follows: Section 2 gives an overview of the broker and its place in the agent environment. Section 3 describes system functions giving explanation for every function in subsections. Section 4 specifies the statistics that the broker gathers. Section 5 describes all the databases used in the broker. Section 6 describes the requirements for handling the debug information. Suggestions for future work are given in section 7.

2 System description

A broker is an agent, whose responsibility is to find a suitable service provider for a requesting agent. The requesting agent don't know the actual service provider agent and therefore the privacy is achieved for both requesters and providers.

A brokered system may be either a pure brokered system or a hybrid system. In a pure brokered system there is only a broker present, which takes care of advertisements, subscriptions and service requests. In a hybrid system there are both broker and matchmaker present. All the advertisements and subscriptions are sent to matchmaker, whereas the broker handles the service requests. The broker and matchmaker cooperate so that the broker may ask the matchmaker for a service provider agents or they may share the information of these so that the broker mirrors the matchmaker's databases of advertisements and subscriptions.

In our broker we are using the DECAF Agent Framework in implementation of the broker agent. Basically the broker is just like any other agent implemented with the DECAF.

2.1 Descriptions of the interfaces

The broker has to register itself to the ANS like any of the other agents. After doing this, it only waits for incoming KQML messages from other agents. Therefore, in a pure brokered system the broker has an interface to the ANS and all other agents. The diagram describing relationships between the broker and other agents in a pure brokered agent architecture are presented in figure 1. The interface to the ANS is not discussed because the DECAF-framework takes care of the registration to the ANS. Descriptions of interfaces between the broker and all “client” agents including the sample KQML messages are presented in the next section.

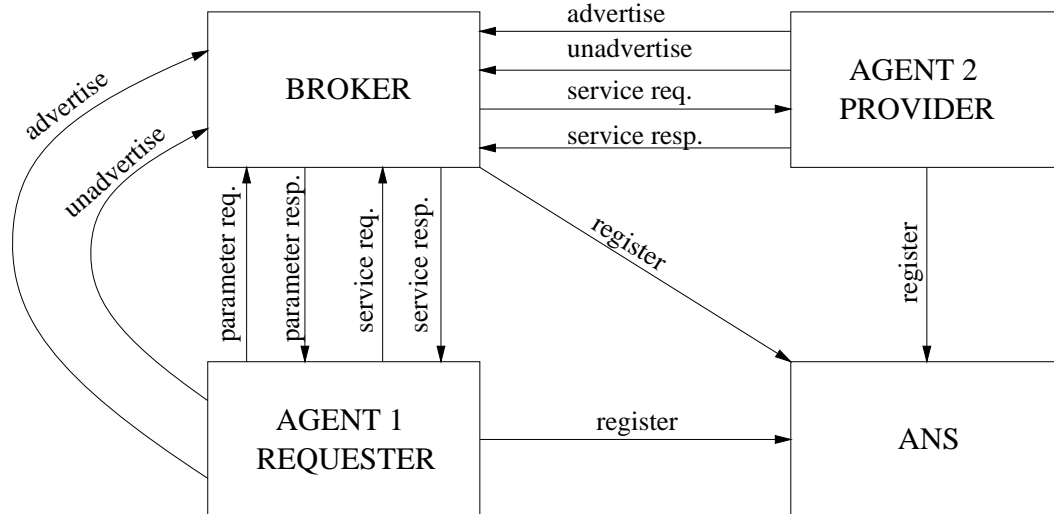


Figure 1: A pure brokered agent architecture.

In figure 2 there are the interfaces of a hybrid agent architecture presented. Basically all the agents advertise themselves for matchmaker, which forwards the advertisements also to the broker. The actual service request go directly to the broker.

3 System functions

In this section all the system functions are described. The subsections describe and illustrate all possible queries and their responses that the broker must be able to handle.

3.1 Advertise

The advertisement is sent first to the matchmaker by an agent who wants to advertise its capabilities to the other agents. The matchmaker then forwards the advertisement to the broker so that the both local databases are up to date. In our case the advertisement is a list of keywords. Keywords are not enough because the requesting agent has to know how to request a service from the providing agent. Therefore, some “instructions” are included about how to request the service. Basically this means that in the advertisement the agent sends also required KQML-fields, that the requester has to use in order to be able to get the desired service. If an

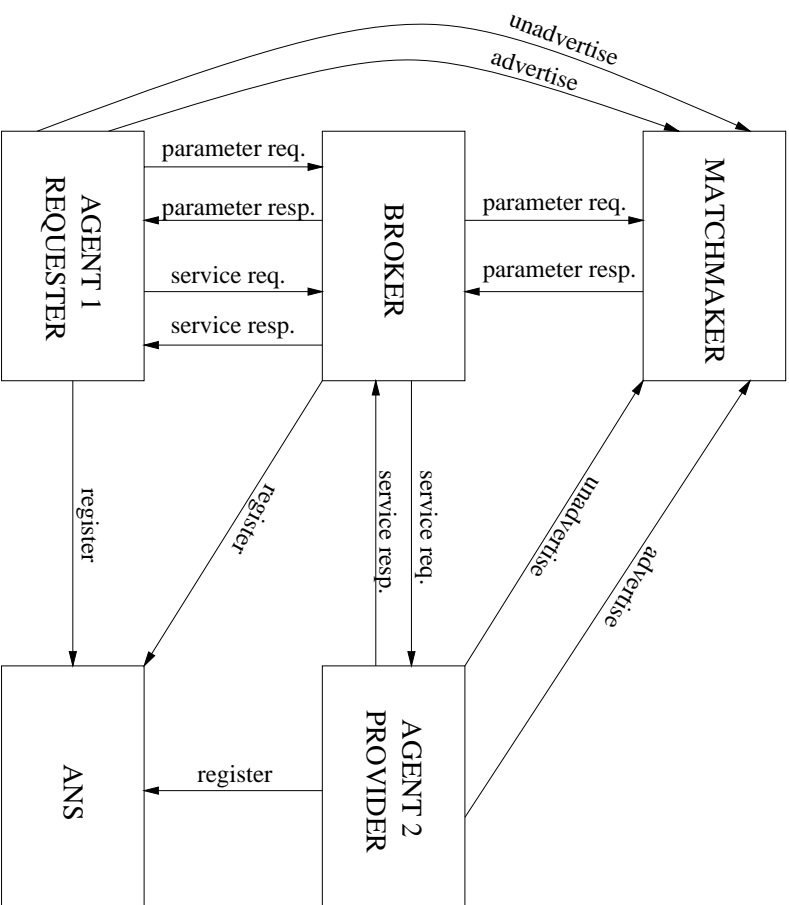


Figure 2: A hybrid agent architecture.

agent wants to advertise multiple actions, it has to do that by sending multiple advertisements.

When the matchmaker receives an advertisement, it stores it into the advertisement database if the advertised action was not in there. If this is the case, then tell-message is sent back to the advertiser. Then the matchmaker checks if the broker is up and running and forwards the advertisement to the broker. If the broker is down, the advertisement is stored only in matchmaker's database. If the advertisement is already in the database, a sorry-message is sent to inform the agent about the failure. Sorry- and tell-messages are specified in section 3.6.

The matchmaker does the matchmaking between the subscription database entries and the keywords of this received advertisement. If a match is found, the corresponding agent(s) from the subscription database is informed. A plan that handles advertisement is presented in figure 3.

When the broker receives the advertisement from the matchmaker, it stores the advertisement into its own advertisement database if the advertised action is not in there. Next the broker does the matchmaking between the subscription database entries and the keywords of this received advertisement. If a match is found, the corresponding agent(s) from the subscription database is informed by sending them a message in which the parameters for asking the service are told.

Finally the broker updates the statistics database by adding a new entry for the new advertising agent in-

serting the name of the agent, current time stamp as the startup time, 0 as service provider load, “up” as the status, 100% as the reliability and 10 as the reliability.

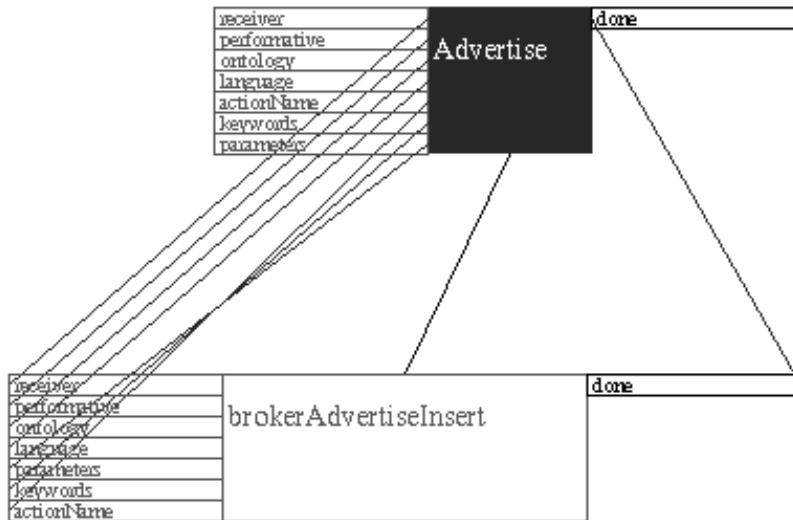


Figure 3: Advertise -plan.

The KQML-message for the advertisement from advertising agent to matchmaker is presented below:

```
(advertise
  :sender      <sending_agent>
  :receiver    Matchmaker
  :ontology    Advertise
  :language    DECAF/Java
  :reply-with  <id_number>
  :content     (:action Advertise
                :receiver <agent_name>
                :performative <performative>
                :ontology <ontology>
                :language <language>
                :actionName <action_name>
                :parameters param_1 param_2 ... param_n
                :keywords word_1 word_2 ... word_n
              )
)
```

The content field contains all the relevant information for the matchmaker and it is stored into the advertisement database.

The KQML-message for the matchmaker to the broker is presented below:

```
(advertise
  :sender      Matchmaker
  :receiver    Broker
  :ontology    Advertise
  :language    DECAF/Java
  :reply-with  <id_number>
  :content     (:action Advertise
                :receiver <agent_name>
                :performative <performative>
                :ontology <ontology>
                :language <language>
                :actionName <action_name>
                :parameters param_1 param_2 ... param_n
                :keywords word_1 word_2 ... word_n
              )
)
```

3.2 Unadvertise

The unadvertisement is used to inform the matchmaker and broker that the agent won't be able to provide the advertised service anymore. The agent can unadvertise one action at a time. When the matchmaker receives an unadvertisement from an agent, it deletes the corresponding entry from the advertisement database. It also checks if the broker is up and running and forwards the unadvertisement to the broker, adding the name of the no longer existing service. The agent is informed by using an untell-message that is presented after the unadvertise-message. The broker also removes the agent's entry as well as the entry from the statistics database, if the agent doesn't offer services anymore.

A plan that handles unadvertisement is presented in figure 4.

The KQML-message for the unadvertisement from unadvertising agent to the matchmaker is presented below:

```
(unadvertise
  :sender      <sending_agent>
  :receiver    Matchmaker
  :ontology    Unadvertise
  :language    DECAF/Java
  :reply-with  <id_number>
  :content     (:action Unadvertise
                :actionName <action_name>
              )
)
```

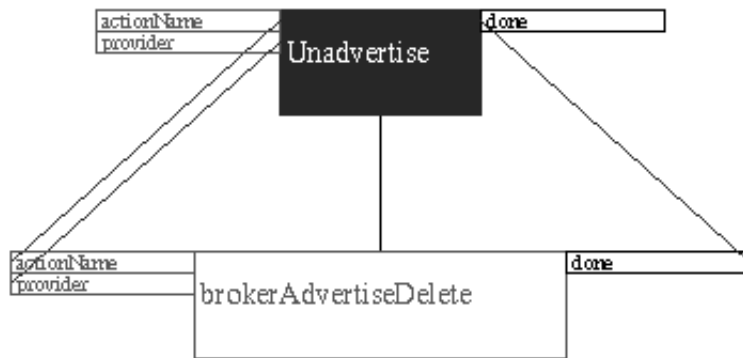


Figure 4: Unadvertise -plan.

The entry corresponding to the agent name (taken from the :sender -field) and the name of the action is deleted.

The KQML-message for the unadvertisement from the matchmaker to the broker is presented below:

```
(unadvertise
  :sender      Matchmaker
  :receiver    Broker
  :ontology    Unadvertise
  :language    DECAF/Java
  :reply-with  <id_number>
  :content     (:action Unadvertise
                :actionName <action_name>
                )
)
```

The KQML-message for untell is constructed as follows:

```
(untell
  :sender      Matchmaker
  :receiver    <requesting_agent>
  :ontology    Unadvertise
  :language    DECAF/Java
  :reply-with  <id_number>
  :in-reply-to <id_number>
  :content     (:action <root_task_for_unadvertisement>
                :actionName <action_name>
                )
  :provider    <provider_name>
```

```
)  
)
```

The agent's root task for unadvertisement is determined by the `:reply-with`-field of the unadvertisement-message.

3.3 Subscribe

Subscribing means that an agent requests to be informed if some specific service comes available on the agent environment. The subscription is presented as a list of keywords as in the normal ask-query. The subscriber determines the quality of matches by giving a threshold percentage that is used in the matchmaking process.

When the matchmaker receives a subscription, it checks if the broker is up and running and forwards the subscription to the broker. The incoming information is stored into the subscription database. After this, the advertisement database is scanned through and checked if there is already an agent that matches the subscription. If there is, the subscriber is informed right a way by using the tell-performative that is explained in the next subsection.

The broker does its own scanning through its own advertisement database, but instead of sending the name of the agent name with the query parameters, it just sends the query parameters. As in asking the actual service, the name of the subscriber agent, query parameters and the name of the service provider agent has to be stored into a cache in order to be able to associate the possible service request from the subscribing agent to the right service provider.

A plan that handles subscription is presented in figure 5.

The KQML-message for the subscription from subscribing agent to the matchmaker is presented below:

```
(subscribe  
  :sender      <sending_agent>  
  :receiver    Matchmaker  
  :ontology    Subscription  
  :language    DECAF/Java  
  :reply-with  <id_number>  
  :content     (:action Subscribe  
                :keywords word_1 word_2 ... word_n  
                :percentage <float>  
              )  
)
```

The KQML-message for the subscription from the matchmaker to the broker is presented below:

```
(subscribe
```

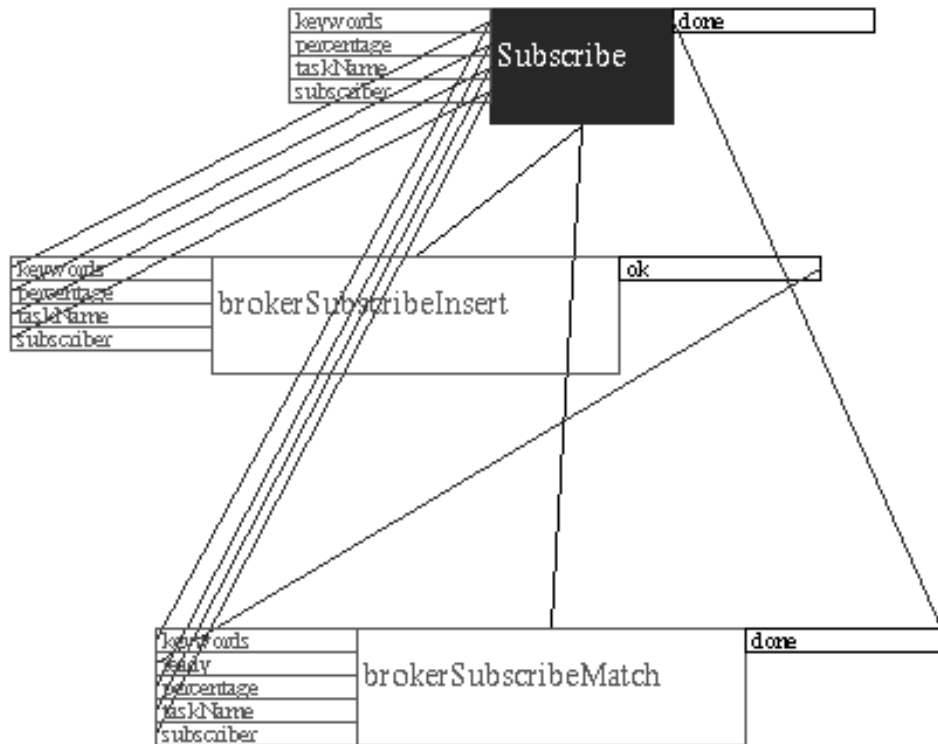


Figure 5: Subscribe -plan.

```

:sender      Matchmaker
:receiver    Broker
:ontology    Subscription
:language    DECAF/Java
:reply-with  <id_number>
:content     (:action Subscribe
              :keywords word_1 word_2 ... word_n
              :percentage <float>
              )
)
  
```

3.4 Unsubscribe

The unsubscription is used to inform the matchmaker and broker that an agent no longer needs to be informed for the subscribed service. Agent can unsubscribe one service at a time. When the matchmaker receives an unsubscription, it deletes the corresponding agent's entry from the subscription database. Then it checks if the

broker is up and running and forwards the unsubscription to the broker with the name of the unsubscriber. The agent is informed of the deletion by an untell-message, which is presented after the unsubscribe-message. When receiving the forwarded unsubscription from the matchmaker, it also deletes the corresponding entry from both its own subscription database and the statistics database.

A plan that handles unsubscription is presented in figure 6.

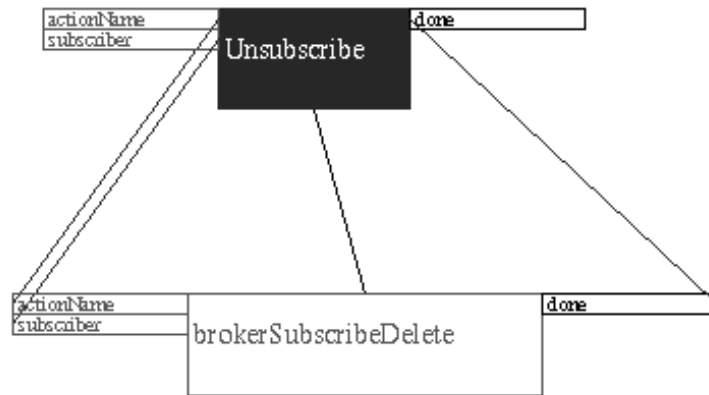


Figure 6: Unsubscribe -plan.

The KQML-message for the unsubscription from unsubscribing agent to the matchmaker is presented below:

```
(unsubscribe
  :sender      <sending_agent>
  :receiver    Matchmaker
  :ontology    Unsubscription
  :language    DECAF/Java
  :reply-with  <id_number>
  :content     (:action Unsubscribe
  :actionName <action_name>)
)
```

The entry corresponding to the agent name (from :sender-field) is deleted.

The KQML-message for the unsubscription from unsubscribing agent to the matchmaker is presented below:

```
(unsubscribe
  :sender      Matchmaker
  :receiver    Broker
```

```

        :ontology      Unsubscription
        :language      DECAF/Java
        :reply-with    <id_number>
        :content       (:action Unsubscribe
                       :actionName <action_name>)
:provider <provider_name>)
)

```

The KQML-message for the confirmation of unsubscription is presented below:

```

(untell
  :sender      Matchmaker
  :receiver    <requesting_agent>
  :ontology    Unsubscription
  :language    DECAF/Java
  :reply-with  <id_number>
  :in-reply-to <id_number>
  :content     (:action <root_task_for_unsubscription>)
)

```

The agent's root task for unsubscription is determined by the `:reply-with` -field of the unsubscription-message.

3.5 Ask, Tell, Sorry

A user agent who wants to get service, sends an ask-query to the broker. The broker does the matchmaking and responds with a set of fields that have to be filled in order to get service from the service provider. The user fills these fields and sends another ask-query to the broker.

First the user agent sends an ask-one -message which includes keywords and the threshold percentage for the match. The broker uses the advertisement database to check if there exists a good enough match. If a match is found, the broker sends the user agent directions of how to request service according to the format specified in the service providers advertisement. The name of the user agent, query parameters and the name of the service provider agent, have to be stored into a cache in order to be able to associate the possible service request from the querying user agent to the right service provider.

The user agent has three choices in requesting service. It can use ask-one, ask-all and stream-all performatives. Ask-one -message is used to request only one answer from the broker. If the user agent wants multiple answers, it can do that with ask-all and stream-all -messages. In the former, KQML messages are sent in a big chunk and in the latter, multiple KQML messages are sent. However, if the service provider is unable to satisfy the request, a sorry-message is sent to the user agent.

The plan that handles the ask-one query is presented in figure 7.

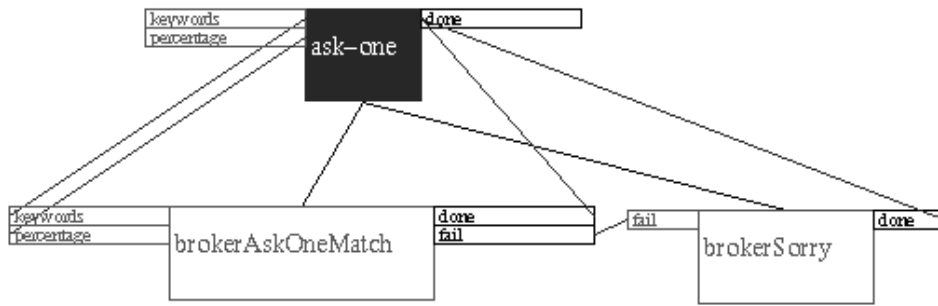


Figure 7: Ask-plan for the broker.

The plan handling service requests is presented in figure 8. Plans taking care of ask-all and stream-all differ only

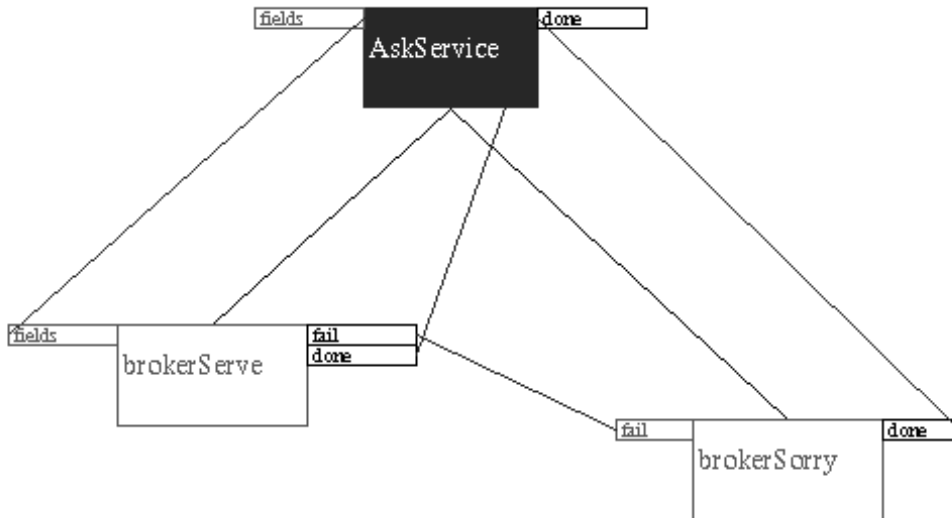


Figure 8: AskService-plan for the broker.

with the name of the root task. Differences that these performatives cause are dealt within the system.

The KQML-message for the ask-one -query for matchmaking is presented below:

```
(ask-one
  :sender      <sending_agent>
  :receiver    Broker
  :ontology    Ask-one
  :language    DECAF/Java
  :reply-with  <id_number>
  :content     (:action Ask
```

```

        :keywords word_1 word_2 ... word_n
        :percentage <float>
    )
)

```

The ask-query for requesting service is presented below:

```

(ask
  :sender      <sending_agent>
  :receiver    Broker
  :ontology    Ask-one
  :language    DECAF/Java
  :reply-with  <id_number>
  :content     (:action AskService
                :fields word_1=value_x word_2=value_y ... word_n=value_n
              )
)

```

The KQML-message for the service request ask-all -query is presented below:

```

(ask-all
  :sender      <sending_agent>
  :receiver    Broker
  :ontology    Ask-all
  :language    DECAF/Java
  :reply-with  <id_number>
  :content     (:action AskService
                :fields word_1=value_x word_2=value_y ... word_n=value_n
              )
)

```

Stream-all differs with ask-all only with the name of the performative and ontology.

The KQML-message for the broker's initial tell-response is presented below:

```

(tell
  :sender      Broker
  :receiver    <requesting_agent>
  :ontology    Tell
  :language    DECAF/Java
  :reply-with  <id_number>
  :content     (:action <user_action>
                )
)

```

The KQML-message for the service request tell-response is presented below:

```
(tell
  :sender      Broker
  :receiver    <requesting_agent>
  :ontology    Tell
  :language    DECAF/Java
  :reply-with  <id_number>
  :in-reply-to <id_number>
  :content     (:action <root_task_name>
                :field_1 <string>
                :field_2 <string>
                ...
                :field_n <string>
              )
)
```

The KQML-message for the sorry-response is presented below:

```
(sorry
  :sender      Broker
  :receiver    <requesting_agent>
  :ontology    Ask
  :language    DECAF/Java
  :reply-with  <id_number>
  :in-reply-to <id_number>
  :content     (:action <root_task_name>)
)
```

The :ontology-field differs according to the performatives that were used. As in tell-message, the :action-field inside the :content-field in the sorry-message is extracted from the :reply-with -field.

4 Statistics

The broker must be able to record and maintain statistics about service providers. Following subsections clarify the type of information that the broker gathers and present the way that the gathering is done.

4.1 Load

Broker measures its own load and the load that is imposed to service providers through user agents.

4.1.1 Broker load

Broker load refers to the burden that requests from service providers and user agents impose on the broker. The burden is measured through the generic load that the computer suffers and the memory that the broker uses on the machine. The load for the machine that the broker is running on is measured by using UNIX's *uptime*-command. The memory that the broker uses is measured by using standard methods provided by Java.

4.1.2 Service provider load

The measurement of load imposed on the service provider and the response time can be used in load balancing. Load is measured in requests the service provider receives per time unit. The number of requests are obtained from the database. The load can then be calculated by dividing the number of requests by the difference between current time and agents startup time.

4.2 Startup time

The startup time of both broker and service providers are taken from the advertisements and recorded so that they can be seen in debug window. The startup time is used when calculating the service provider load.

4.3 Status

Service providers can be in three different stages: up, not responding or sleeping. Sleeping will be implemented in the future. The agent is considered to be up when it has advertised and not responding if it doesn't respond to an ask request within a specified time frame. If the agent crashes or does a shutdown, it should send an unadvertisement to the matchmaker which will forward it to the broker. When broker receives the unadvertisement, it will remove the agent from its database. If the agent fails to send the unadvertisement, its status will remain *not responding* until it can be considered to be down and removed from the database of the broker.

4.4 Response time

The broker will measure response time of service provider agents by calculating the difference between sending an ask request and receiving a response.

4.5 Reliability

The reliability of an agent is the ratio of answers (other than sorry) to all questions.

4.6 Quality

User agents are encouraged to give feedback on the quality of the answers that service provider agents. This information is recorded in statistics database of the broker and used as a factor in choosing service providers.

5 Databases

The broker has three databases: advertisement database, subscription database and statistics database. The advertisement and subscription databases are specified in the specification of the DECAF-matchmaker. [7].

5.1 Database for the statistics

The broker maintains statistics about the agents that are advertised themselves with the matchmaker and broker as well as statistics of its own functionality. Entries in the database contain agent name, service provider load (stored as number of requests), startup time, status, response time, number of OK replies, number of Sorry-replies and quality. The database will be implemented as an ascii file, where a line corresponds to an entry. The separation mark between the fields in an entry is “;” and the separation mark between the keywords and the parameters is one space. All the other fields are represented as a string without spaces. Example of the database entry is shown below:

```
Agent1;16;09:38;up;2.6;12;3;9
```

6 Debug information

The debug information is handled by a separate debug class. If the debugging is turned on, the debug class is created and all the debugging information is forwarded to the debug class that takes care of showing it in a separate window. The debug information is separated from the actual implementation of the broker to ease further development and to clarify code. The debug part can be either removed, improved or replaced altogether without touching the broker agent. In fact in future releases of the DECAF-framework the handling of the debugging information may be improved so that a debug information specific to the broker is not needed at all.

The debug information is divided into four parts. The first panel shows the advertisements. The second panel presents all the subscriptions. Third panel shows statistics about the broker and registered service providers. The fourth panel is a generic debug window which shows all the information that has something to do with the actions the broker performs. All the panels are scrollable. The user can see e.g. the incoming and outgoing KQML-messages as well as the results from the matching process from the generic debug panel. The debug window is presented in figure 9.

7 Future work

TODO...

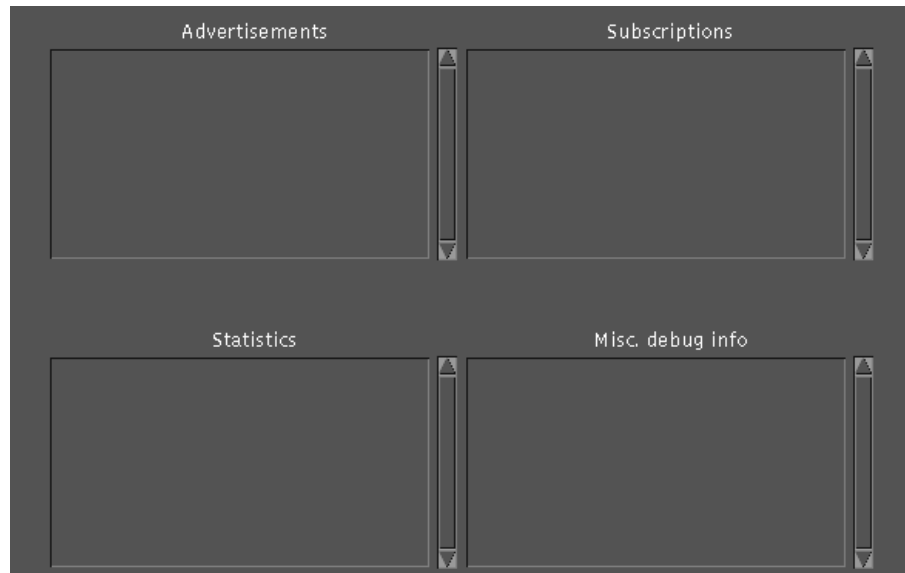


Figure 9: The debug window.

References

- [1] K. Sycara, J. Lu, M. Klusch, S. Widoff. Matchmaking among Heterogeneous Agents on the Internet
- [2] K. Decker, K. Sycara. Intelligent Adaptive Information Agents. *Journal of Intelligent Information Systems*, 9, pp. 239-260, 1997
- [3] K. Decker, K. Sycara, M. Williamson. Modeling Information Agents: Advertisements, Organizational Roles, and Dynamic Behavior
- [4] K. Decker, A. Pannu, K. Sycara, M. Williamson. Designing Behaviors for Information Agents
- [5] K. Decker, K. Sycara, M. Williamson. Middle-Agents for the Internet
- [6] Leonard N. Foner. A Multi-Agent Referral System for Matchmaking
- [7] M. Laukkanen, J. Eskelinen. Requirement specification for the DECAF-Matchmaker