OpenKnowledge

FP6-027253

OpenKnowledge File Front-End

David Dupplaw[1], Paolo Besana[2], Madalina Croitoru[1], Srinandan Dasmahapatra[1], Bo Hu[1], Paul Lewis[1], Antonis Loizou[1], Liang Xiao[1]

[1] IAM Group, School of Electronics and Computer Science, University of Southampton, Southampton, SO17 1BJ, UK.
[2] Centre for Intelligent Systems and their Applications, University of Edinburgh, Edinburgh, EH8 9LE, UK.

Report Version: final
Report Preparation Date: December 2007
Classification: deliverable 5.4
Contract Start Date: 1.1.2006 Duration: 36 months
Project Co-ordinator: University of Edinburgh (David Robertson)
Partners: IIIA(CSIC) Barcelona
Vrije Universiteit Amsterdam
University of Edinburgh
KMI, Open University
University of Southampton
University of Trento

OpenKnowledge File Front-End: Using Interaction Deferral and Delegation

David Dupplaw Paolo Besana Madalina Croitoru
Srinandan Dasmahapatra Bo Hu Paul Lewis
Antonis Loizou Liang Xiao

January 2, 2008

Abstract

OpenKnowledge aspires to create networks of peers that are able to share data. This deliverable describes the file-front end that allows peers to share larger data items than would be sensible to share during an interaction. It is implemented as an OpenKnowledge component. It is described in the context of a multimedia application and introduces the idea of deferred interaction model execution. For deferred interaction model execution, the subscription negotiators, that provide the list of mutually compatible peers during interaction bootstrapping, need to be able to select peers by the specific identifiers.

1 Introduction and motivation

The well-known peer-to-peer networks, like Kazaa, eMule, etc., provide the means for users to share files. Although this is not the aspiration of the OpenKnowledge project, providing a means for sharing data is an important part of allowing peers to interact. The interaction of peers is controlled by a coordination language, LCC [3, 4], and small data items can be transferred between peers using the standard OpenKnowledge messaging systems of the peers during a normal interaction. However, there are many data items that would be too large to be sensibly passed in interaction messages due to the time overhead transferring them would incur. We may wish to first make available cheap summary metadata about the data with an option for the requesting peer to retrieve the full data should they require it.

We have previously presented a means for large data items to be described with summary metadata descriptions that can be transferred using the standard messaging system [2]. Within the interaction these metadata objects are stored as the value of a single variable but represent the entire object. The aspiration is that the transfer of the large data object can be deferred until the data is required. The challenge is for the OpenKnowledge peer to transfer the data transparently to minimise the authoring overhead for interaction models.

The file front-end is a small protocol implementation that allows files to be intentionally shared on the network by users but also allows running interactions, that involve large data items, to share data transparently. The implementation is realised, naturally, as an LCC interaction model with supporting OpenKnowledge components. The protocol itself is a very simple bilateral interaction but utilises two interesting parts of the OpenKnowledge kernel: interaction model deferral and delegation and subscription negotiation.

To make concrete the ideas that are explained here we will introduce a very simple application that will utilise the file-front end.

2 The Photo-Library Scenario

A magazine editor has OpenKnowledge running on her peer and wishes to buy a photograph for the front-page of her special-issue magazine all about insects. She needs to initiate an interaction with a photographic library that allows photographs to be purchased. The magazine editor subscribes to an appropriate interaction model that she found already published on the network. She subscribes in the *photo_buyer* role and starts an interaction with a photographic library who is already subscribed in the *photo_lib* role. The editor enters the keyword 'insects' and is returned a list of possible images that she is able to interrogate based on their licencing conditions, overall resolution, relevancy, cost and other metadata provided by the library. She chooses the one she thinks is perfect for the front page of her magazine and proceeds with the purchase. Her payment is processed without a problem and the image is downloaded to the editor's peer and shown to her. As it is now on her peer, she can import it into her design application just like any other file.

The interaction model the editor and the photo library subscribed to is shown in Model 1.

$$
\begin{array}{l}
\mathbf{a}(\text{photo\_lib}, PL) :: \\
\quad \mathit{photoRequest}(\mathit{Keywords}) \Leftarrow \mathbf{a}(\text{photo\_buyer}, PR)\ \mathit{then} \\
\quad \mathit{photoDetails}(\mathit{ImageList}) \Rightarrow \mathbf{a}(\text{photo\_buyer}, PR) \\
\qquad\qquad \leftarrow \mathbf{getImages}(\mathit{Keywords}, \mathit{ImageList})\ \mathit{then} \\
\quad (\\
\qquad \mathit{buyImage}(I) \Leftarrow \mathbf{a}(\text{photo\_buyer}, PR) \leftarrow \mathbf{inList}(\mathit{ImageList}, I)\ \mathit{then} \\
\qquad \mathit{imageBought}(I) \Rightarrow \mathbf{a}(\text{photo\_buyer}, PR) \leftarrow \mathbf{acceptPayment}(PR)\ \mathit{or} \\
\qquad \mathit{paymentFailed}(I) \Rightarrow \mathbf{a}(\text{photo\_buyer}, PR) \\
\quad)\ \mathit{or} \\
\qquad \mathit{cancel}() \Leftarrow \mathbf{a}(\text{photo\_buyer}, PR) \\
\\
\mathbf{a}(\text{photo\_buyer}, PR) :: \\
\quad \mathit{photoRequest}(\mathit{Keywords}) \Rightarrow \mathbf{a}(\text{photo\_lib}, PL) \\
\qquad\qquad \leftarrow \mathbf{getKeywords}(\mathit{Keywords})\ \mathit{then} \\
\quad \mathit{photoDetails}(\mathit{ImageList}) \Leftarrow \mathbf{a}(\text{photo\_lib}, PR)\ \mathit{then} \\
\quad (\\
\qquad \mathit{buyImage}(I) \Rightarrow \mathbf{a}(\text{photo\_lib}, PL) \leftarrow \mathbf{selectImage}(\mathit{ImageList}, I)\ \mathit{then} \\
\qquad (\\
\qquad\quad \mathit{imageBought}(I) \Leftarrow \mathbf{a}(\text{photo\_lib}, PL)\ \mathit{then} \\
\qquad\quad \mathit{null} \leftarrow \mathbf{getImage}(I, I_{data})\ \mathit{and}\ \mathbf{saveImage}(I_{data})\ \mathit{then} \\
\qquad\quad \mathit{null} \leftarrow \mathbf{showImage}(I, I_{data}) \\
\qquad)\ \mathit{or} \\
\qquad\quad \mathit{paymentFailed}(I) \Leftarrow \mathbf{a}(\text{photo\_lib}, PL) \\
\quad)\ \mathit{or} \\
\qquad \mathit{cancel}() \Rightarrow \mathbf{a}(\text{photo\_lib}, PL) \\
\mathbf{visual}(\mathbf{getKeywords}(\mathit{Keywords}), \mathbf{input}(\text{“Enter Search Terms:”}, \mathit{Keywords})) \\
\mathbf{visual}(\mathbf{selectImage}(\mathit{ImageList}, I), \mathbf{chooseImage}(\mathit{ImageList}, I)) \\
\mathbf{visual}(\mathbf{showImage}(I, I_{data}), \mathbf{showImage}(I_{data}))
\end{array}
\tag{1}
$$

Model 1, the model for the photographic library application, is based on a standard request-response pattern.

In the initial search results list, the server initially returns only metadata about the requested photograph to the requester; for example, it may return the licencing conditions, the price, the image resolution, creator, etc. This not only makes sense from a efficiency point-of-view, but also makes particular sense in this case, as a photo library would not want to send the full-resolution photograph to the buyer before they had paid for it. This metadata will be in a language that the photo library has considered to be best for them. However, it may not match the requester's local knowledge representation, so mapping between schemas will need to take place for execution of the process.

Once a user selects an image to buy, the payment is processed and the server responds with an $\mathit{imageBought}(I)$ message, where I contains the image metadata (that may undergo mapping).

The *getImage* constraint of the *photo_buyer* role is where the interaction model delegation takes place before the image is saved to the disk. The **getImage**(I, I_{data}) retrieves the image data from the photo library based on the implicit image information, in I, that is provided by the photo library. The type and value of I is deliberately tacit; the open nature of the interaction model definition should not be constrained to any particular image format. It is therefore imperative that the mapping is able to provide good enough results for implementations of the **getImage**(I, I_{data}) method. So, clearly, the way the image-data retrieval is actually achieved depends on the OKC implementation. In the interest of showing the interaction deferral and delegation functionality we can assume here that the retrieval of the image data, I_{data}, uses an interaction model that the photo library server has provided to the buyer through the image metadata, I, in the message $imageBought(I)$. The interaction model defines an interaction that will allow the buyer to retrieve the actual image data of the photograph. When the buyer requires the actual image data they can invoke the interaction model included with the image metadata which will start an interaction with the photo library and download the image to their peer. This interaction model is an instance of the file-front end and an example of the interaction deferral which we will describe in the following sections.

3 File Front-End Requester Component

The file front-end *requester* role (the role the *buyer* will delegate to in our scenario) is a simple protocol that states a peer will make a request for a given data item from a peer in the *supplier* role, when it knows the item identifier. The peer in the *supplier* role then returns the data item's raw data, or returns an error message. On success, the data is stored onto the local peer. In Section 7 we will describe how this model can provide other means for data transfer.

The LCC model is shown in Model 2.

$$
\begin{array}{l}
\mathbf{a}(\text{ffe\_requester(ItemID,Data,D)}, R) :: \\
\quad retrieve(ItemID) \;\Rightarrow\; \mathbf{a}(\text{ffe\_supplier}, D) \\
\quad dataItem(Data) \;\Leftarrow\; \mathbf{a}(\text{ffe\_supplier}, D) \leftarrow \mathbf{saveData}(Data) \\
\quad or \\
\quad error(Msg) \;\Leftarrow\; \mathbf{a}(\text{ffe\_supplier}, D)
\end{array}
\tag{2}
$$

It is clear how a user might use this to manually request items from other peers, but by statically injecting the values for $ItemID$ and/or D into the interaction model, this model will only perform one single specific request. This makes the application of the interaction model delegation easier to handle for the requester.

In the photo library example, the item returned in the *imageBought* message is a metadata structure that contains an instance of this interaction model. The instance will have some statically injected values that limit the requester to retrieving only a certain photograph from a certain photographic library.

Model 3 shows how the input to the model might be statically defined for an item by the photo library server.

$$\begin{array}{l}\mathbf{a}(\text{ffe\_requester}(\text{“12345”},\text{Data},\text{“photolib.com:4000:AB43AB3”}),R) :: \\ \quad \ldots\end{array} \tag{3}$$

If this interaction model was executed the requester would only communicate with the server component at the given *EndPointID* (photolib.com:4000:AB43AB3) and only for the given item identifier (12345). Doing so allows the requester to be agnostic to the means for retrieval of the item and allows the server to limit, to some extent, the actions of the requester.

When the requester subscribes to this model, a bootstrapping mode is entered that allows the supplier to check which requesters are wanting to download data. The *Subscription Negotiator* module of the kernel on the photo library peer handles this bootstrapping interaction, as is described below with the complimentary part of the file front-end model.

4 File Front-End Delivery Component

The file front-end delivery component is the complimentary protocol to the requester protocol and so is also very simple. It delivers some raw data when requested, or an error message if the data is not available.

Model 4 shows the file front-end delivery protocol. This model accepts a request for a data item, identified by a given item description that may need to be mapped to the delivery peer's local ontology. If the item exists and the instantiated role is able to associate it as the value of a variable, the role returns the item in the message $dataItem(Data)$.

$$\begin{array}{l}\mathbf{a}(\text{ffe\_supplier}, FFE) :: \\ \quad null \leftarrow \mathbf{getItemToShare}(Item)\ and\ \mathbf{shareItem}(Item) \\ \quad \mathbf{a}(\text{ffe\_supplier(Item)}, FFE) \\ \mathbf{visual}(\mathbf{getItemToShare}(Item), \mathbf{chooseFile}(Item))\end{array}$$

$$\begin{array}{l}\mathbf{a}(\text{ffe\_supplier(Item)}, FFE) :: \\ \quad retrieve(Item) \Leftarrow \mathbf{a}(\text{ffe\_requester}, R) \leftarrow \mathbf{exists}(Item) \\ \quad dataItem(Data) \Rightarrow \mathbf{a}(\text{ffe\_requester}, R) \leftarrow \mathbf{getItem}(Item, Data) \\ \quad or \\ \quad error(Msg) \Rightarrow \mathbf{a}(\text{ffe\_requester}, R) \leftarrow \mathbf{getLastError}(Msg)\end{array} \tag{4}$$

A user could manually subscribe to this interaction model directly using the first of the clauses. This allows the user to select an item from their local disk and flag it in the peer's state as being shareable. Alternatively, an actor can begin execution at the second clause with an item ID pre-defined, as would

occur if the data were being delivered as part of a deferred interaction model execution. In this case, it must be ensured that the item data is only delivered to the correct requester and instantiation of an appropriate subscription negotiator at the deliverer can help to achieve this.

5 Subscription Negotiation

The subscription negotiator is a module that exists for each subscription on every peer and is responsible for deciding on which other peers it is prepared to play with in that interaction.

During interaction bootstrapping, the *coordinator* receives from the *Discovery Service* a list of all the possible actors that may play the roles in the interaction model that the coordinator has been selected to orchestrate. The coordinator asks each peer for a list of peers that it is prepared to play with for each role. The coordinator then runs a constraint satisfaction algorithm on the result to find the maxmimum set of mutually compatible peers. If all roles are filled by at least the minimum number of peers that the interaction model defines, the interaction is started.

All OpenKnowledge Components (OKCs) have end-point identifiers that allow a specific OpenKnowledge component to be identified uniquely on the network. For OKCs that are playing in an interaction, identity of the other players is provided by the coordinator. End-point identifiers consist of two parts: one, a peer identifier which in the current kernel implementation is the IP address of the machine with a port number but theoretically can be any unique identifier; and two, a unique identifier that represents the OKC on that peer. During creation of a subscription negotiator, the server must ensure that the comparison routine will match peer identifiers and not end point identifiers, as the OKC that will play on the client side in the deferred interaction will not be known in advance, but the peer will. For a computer on which multiple peers are running, such as a web-based OpenKnowledge server, each peer identifier should be associated with a single user, thereby making the peer identifier synonymous with a user.

The kernel is able to provide a subscription negotiator that will only choose a specific peer for each role. The OKC generating the deferral will ask the peer to create a subscription for a given interaction model. The peer is the central knowledgeable unit in the kernel and all peer processes must be handled via the peer. Therefore, the OKC asks the peer to subscribe, although the peer is not obliged to do so. Listing 1 shows the Java code for subscribing to an interaction model.

The *subscribe* function that is provided by the OKC will ask the peer to subscribe to the given model in the given role (with the given arguments). The returned negotiator is set to only accept the peer identified by $peerID$ in the role $ffe\_requester$. Both $peerID$ and $Item$ are local to the OKC in the peer and will be gathered from the interaction variables.

An extension to this mechanism, that will make the subscription process

```
if( publish( lcc ) )
  logger.warn(''Could not publish'');

SubscriptionNegotiator sa =
    subscribe( lcc,''ffe_supplier'',Item );
if( sa == null )
  logger.warn(''Could not subscribe'');
else
  sa.addFilter( new PeerIDFilter(peerID,
    new Role(''ffe_requester'')) );
```

Listing 1: OKC Code for interaction subscription

more efficient, is to ensure that the discovery service filters the potential actors for an interaction prior to negotiation. If the kernel and discovery service were able to provide a form of tagging for peers, groups can be formed within the peers such that peers that are to play in a closed interaction can be pre-selected by the discovery service using their group tags. Doing so will mean the number of peers that are contacted during bootstrapping will be greatly reduced. However, the subscription negotiation phase still need to be robust such that malicious peers that are tagging themselves incorrectly will still be disallowed from playing in the interaction.

6 Interaction Delegation

In an interaction, a peer may receive a metadata item that contains a deferred interaction model that allows retrieval of the actual data. The peer must block its execution of the current interaction and delegate the execution to the other interaction model instance. In this way, the interaction delegation process is a means for a constraint satisfaction routine to depend on the result of the execution of another interaction model. This mechanism provides a powerful way to make the peers more proactive.

OpenKnowledge components are modules that are used to satisfy constraints in interaction models. In the OpenKnowledge Java SDK, these components should extend the *OKCFacadeImpl* class that provides a limited set of functions that the developer is able to use to communicate with the kernel on which the OKC is running. It is in this abstract class that we add a function that allows the OKC to subscribe the peer to other interactions.

The deferred interaction model definition is provided in LCC, along with the identifier of the role which the peer should play when delegation takes place. The deferral should have a flag that informs the peer whether another peer will already be subscribed in a role on that interaction model. In the case of the file front-end, another peer will already be subscribed in the $ffe\_delivery$ role when the interaction model is received by the requester. This means the

receiving peer is able to simply subscribe to the interaction. If this flag indicates that the actor list may be entirely distinct from the list of actors in the current interaction model, the peer is obliged to first publish the interaction model, as it may not exist on the network. Publishing models that are already published has no effect in the OpenKnowledge network.

Allowing OKCs to subscribe the peer to interactions also requires the OKC to specify new OKCs that will be used to solve constraints in that new interaction. As this is usually a manual step that the user initiates, we need an intelligent way in which to achieve the association of OKCs with the interaction instance.

There are two parts to the procurement of an appropriate OKC for subscription to a deferred interaction model. The most obvious is that the local peer's functionality (OKC repository) is searched for appropriate components that will play in the interaction. Components that have a direct mapping may be used without user intervention, however if a mapping takes place to ensure interoperability, the user may be required to validate the OKC. The 'Good Enough Answers' score can be used to determine this.

In the case that an appropriate OKC is not found in the local peer's functionality, it could be procured from the network, if the user's policy allows this. As every interaction model has a unique identifier, OKCs that are associated with that identifier directly, could be located on the network and downloaded (with the appropriate permission from the user). These could then be instantiated and used in the new interation instance. Finding appropriate OKCs on the network that are not directly associated with interaction models has not been implemented in OpenKnowledge.

Once the appropriate OKCs exist on the local peer, they must be instantiated as part of a subscription negotiator, and the subscription negotiator should, again, be aligned to the peer from which the deferred interaction model was received. Once the subscription is complete, the interaction model will play out until it ends and control is returned to the OKC that subscribed to the model.

In LCC, roles within interaction models do not have return values. However, parameterised roles can be used to identify return values, by side-affecting the parameter values, as long as these roles are used as entry roles. The call to the parameterised role, in the OKC initiating the delegation, would have to align with the parameterised role in the deferred interaction model definition such that the values of the return variables can be copied into the variables of the blocked interaction model. Indeed, a subscription to the parameterised role could automatically transfer the required values to the OKC's scope to make the process easier to implement.

For example, a subscription to the parameterised role $ffe\_requester$, in Model 2 would use the OKC's in-built functions to initiate the subscription, while automatically updating the local symbol table at the end of the delegation. The listing below shows a simplified call to the subscription function, where $Item$ and D are local variables in the blocked OKC.

Clearly, in the case that $ItemID$ is statically defined in the model, the initial value of $Item$ will have no affect, but it will be side-affected at the end of execution of the $subscribeAndWait$ function call.

```
if( !subscribeAndWait(lcc,''ffe_requester'',Item,Data) )
  logger.warn(''Could not subscribe'');
```

Listing 2: OKC code for interaction delegation

7 On-The-Fly OKC Installation

It could be that the raw data that the client requires might be better delivered using a different technique than the standard OpenKnowledge messaging. For example, if the user required a large video, it may be more efficient that the video is retrieved from many peers all at once as the BitTorrent protocol provides; or an application server may be maximising the control of their data by using a proprietry communication method. The default transfer mechanism as provided by the OpenKnowledge messaging system may not be enough for all types of data and it is in the interest of an open system that OpenKnowledge provides other options for data transfer. The definition of the file-front-end is generic enough that other mechanisms can be used if OKCs are provided that implement the constraints in a different manner.

Model 2 introduced the interaction protocol for the requester of data. The model defines the constraint **saveData**(*Data*) that stores raw data that has been transferred from the server, onto the local peer.

However, it could be that the implementation of the **saveData**(*Data*) constraint uses BitTorrent to download the data, whose reference is given by the server in the variable *Data*. It could be that the implementation uses a proprietory method for downloading the data and needs to enact a specific low-level protocol that is based on some information in the value of *Data*. Either way, the implementation is specific to the retrieval technique and may be in a form that the client does not yet know about.

To overcome this, the kernel can provide a mechanism by which OKCs are installed on-the-fly into the local peer. The server should publish the relevant OKCs to the network and pass, in the image metadata, the relevant OKC identifier such that the client can retrieve the OKC from the network and install it. Alternatively a more flexible approach could be used, by mapping the semantic markup.of the data to the semantic markup of OKCs on the network.

For the appropriate OKC to be identified on-the-fly by the kernel, when no OKC identifier is provided, the kernel must be able to identify the protocol for data retrieval and use it in the mapping process for subscription. If no match can be found for particular constraints in the subscription, the appropriate OKC can be sourced from the network based on its semantic markup. As long as the OKC code is semantically marked up (as described in Deliverable 1.3 [1]) the process can be automated.

Clearly, downloading and installing OKCs on-the-fly could introduce some security concerns. It is therefore essential that the user is informed of such on-the-fly installations and be responsible for giving the go-ahead to download an install the appropriate OKCs.

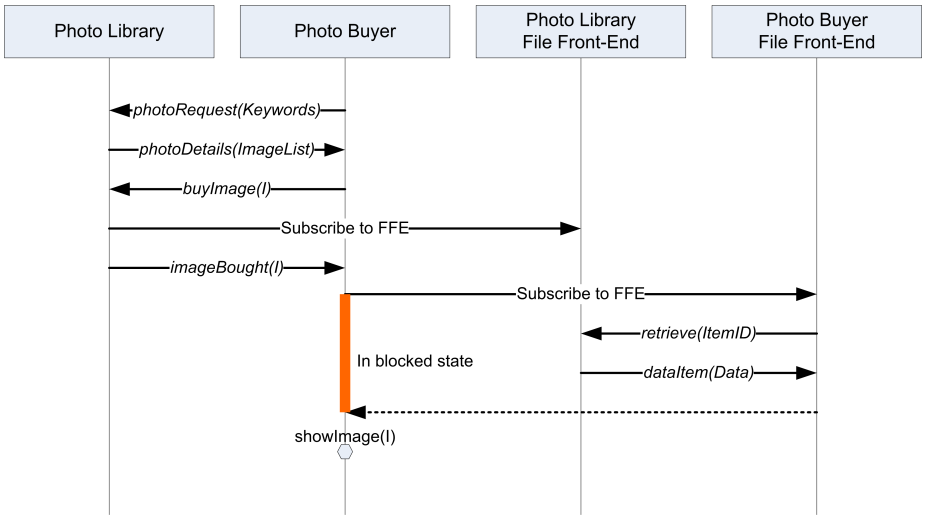

Figure 1: The Photo Library deferred interaction model sequence diagram

8 The Photo Library Deferral and Delegation Process

If we return to the photography library example, we can show how the deferral process works in context. Figure 1 shows the sequence of messages in the interaction, showing the OKCs and their interactions.

Initially, the interaction takes place as normal. The photographic library returns a set of image metadata based on the user's keyword terms. When the user successfully pays for an image, the photo library server subscribes to the file front-end delivery component with buyer specified in its subscription negotiator's role filter. When the buyer received the *imageBought* message, containing the deferred interaction model definition, it is able to begin the execution of that interaction model (delegate) to retrieve the image data. It subscribes to the file front-end model and bootstrapping begins. The server's subscription negotiator for the OKC that will deliver the bought image, will only accept the buyer (or their peer identifier) to play with in the requester role so only the buyer is able to receive the requested image. When the delegated interaction completes the original interaction is able to continue - in this case, the *showImage* constraint is satisfied.

9 Conclusions

In this article, we have discussed a simple OpenKnowledge component that is able to deliver data between peers. We have introduced how the component can be provided as a deferred interaction model for retrieval of large data, thereby

allowing the message passing to be efficient and responsive in cases where the large data transfer would be unnecessary. We described how the deferred model can be executed using the interaction model delegation mechanism that blocks the execution of an interaction model and delegates execution to another model instance. The results of a delegation can then be used when the blocked model restarts. To achieve the communication between the peer that generates and subscribes to the deferred model and the peer that receives and subscribes to the deferred model, the kernel's subscription negotiator needs to be extended to allow selection of specific peer identifiers in specific roles. The kernel's component APi also needs to be carefully modified to allow OKCs to subscribe to roles on the networkm both asynchronously and synchronously. Implementing these changes in the kernel will provide much greater autonomy for peers and should provide some powerful functionality.

References

[1] Paolo Besana, *OpenKnowledge plug-in components*, Tech. Report Deliverable 1.3.

[2] Antonis Loizou, Mischa M. Tuffield, Paul H. Lewis, David Dupplaw, Madalina Croitoru, Liang Xiao, and Srinandan Dasmahapatra, *Markup tools for OpenKnowledge*, Tech. Report Deliverable 5.2.

[3] Dave Robertson, *A lightweight coordination calculus for agent systems*, Declarative Agent Languages and Technologies, 2004, pp. 183–197.

[4] David Robertson, *Multi-agent coordination as distributed logic programming.*, ICLP, 2004, pp. 416–430.