AMIDA
Augmented Multi-party Interaction with Distance Access

http://www.amidaproject.org/

Integrated Project IST–033812

Funded under 6th FWP (Sixth Framework Programme)

Action Line: IST-2005-2.5.7 Multimodal interfaces

Deliverable D7.2: Commercial Component Definition

Due date: 30 November 2007
Project start date: 1/10/2006
Lead Contractor: PHI

Submission date: 30 November 2007
Duration: 36 months
Revision: 1.0

Project co-funded by the European Commission in the 6th Framework Programme (2002-2006)		
Dissemination Level		
PU	Public	✓
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

D7.2: Commercial Component Definition

Abstract:

The AMIDA project is developing technology to assist meeting participants, particularly during "remote" meetings conducted using videoconferencing and other communication tools. Our approach relies on employing techniques for automatically analysing meetings either as they happen or after the fact and allowing end user applications access to the "annotations" that have been derived from the raw signals. In this first year of the project, we have designed and implemented the infrastructure for handling these annotations. This report describes this infrastructure, called "the Hub", and how it has been used to broker between components that produce annotations and demonstrations of potential end user applications.

Contents

1 INTRODUCTION 4
2 THE OVERALL CONCEPT OF THE HUB 5
3 THE OVERALL DESIGN 6
3.1 PRODUCERS 6
3.2 CONSUMERS 7
3.3 DATA AND METADATA MODEL 7
3.4 ANNOTATION ARCHIVE (TRIPLE STORE) AND HUB BASE 8
3.5 MIDDLEWARE 9
3.6 RUNNING DEMONSTRATIONS AND ATTACHING NON-JAVA PRODUCERS 9
4 HOW TO USE THE HUB AND MIDDLEWARE 9
4.1 CONNECTING TO THE HUB 10
4.2 EXAMPLE CONSUMER 10
4.3 EXAMPLE PRODUCER 11
4.4 EXAMPLE METADATA QUERY 11
4.5 PACKAGE STRUCTURE 12
5 UNDERLYING TRIPLE REPRESENTATION 12
5.1 DATA ELEMENTS 12
5.2 METADATA ELEMENTS 13
6 DISCUSSION 14

1 Introduction

The AMIDA project is developing technology to assist meeting participants, particularly during "remote" meetings conducted using videoconferencing and other communication tools. We are focusing on two different types of target applications that we think will be the most valuable. The first is *engagement support for remote meetings*. This involves techniques to help the user overcome the special difficulties that arise from taking part in a meeting remotely. When people are face-to-face, they have many subtle cues available to them about everything from whether or not the other person is paying attention and if so, to what; what they mean by what they say; and how they feel about what is happening. These cues are often missing during remote meetings, even when using relatively rich interfaces like low-latency videoconferencing. Our goal for this kind of application is make substitutes for at least the most basic of these cues available to remote meeting participants, in order to improve their meeting experiences. The second type of target application is *meeting information access.* During meetings, participants often need to remember information relevant to the current discussion, whether this information is in a formal document, in their email, was expressed in the minutes of previous meeting, or was just discussed previously without being minuted. Our goal for this kind of application is to produce an interface that meeting participants can use that will "listen in" during an ongoing meeting an unobtrusively pull up the most relevant information it has available, to reduce the amount of time users spend searching for it. Although participants in all meetings could potentially benefit from such an application, it is remote meeting participants who have the most to gain, since they need to concentrate more of their attention than face-to-face ones on the meeting at hand simply to understand what is happening.

Applications such as these rely on the automatic analysis of meeting recordings for the properties that will affect the state of the end user application. For instance, for engagement support, if a remote participant is joining a meeting that is largely face-to-face, it might be useful to convey roughly where each of the face-to-face participants is looking. This might tell the remote participant, for instance, whether there is crucial information on the slides or on a physical document they do not have, or whether everyone else is currently distracted. For meeting information access, knowing which past information is relevant requires an analysis of the words that have been spoken, not just in the current meeting, but also in past meetings, so that the correct past extracts can be found. There is the potential for a great many different types of automatic analysis, producing different annotations, to be performed on the same base multimodal signals. There could also be many different end user applications requiring the same annotations, even at the same time; consider the case where there are two simultaneous meetings both of which need to remember a critical event in the company's past. Although we could treat every application separately, producing the annotations and then accessing them, this would be highly inefficient. Instead, we have devised an annotation infrastructure through which components that produce annotations can bank what they produce, and components that need the annotations can retrieve them. We call this infrastructure "the Hub". The main documentation for the Hub is the Javadoc accompanying its implementation; this document introduces the system at a higher level.

2 The Overall Concept of the Hub

The Hub is intended to provide all of the storage that a group or a company needs for annotations about their archived meetings in one place. Our vision for how this would work is as follows. Once a company decides that it wishes to archive its meetings, it would purchase a Hub, much like it currently might purchase a videoconferencing system or any other major piece of infrastructure. Installing the Hub would involve setting up a Hub server to run the database in which the annotations are stored, a method for backing it up, and possibly redundant servers to spread the processing load and reduce the times the overall system is down for maintenance. It would also require the company's meeting rooms and videoconferencing end points to have recording equipment set up, in the way that the AMI and AMIDA projects have demonstrated, with the audio and video signals going into a separate time-synchronized signal store. These signals would feed straight into processes for creating annotations automatically from them: automatic speech recognition to yield the words spoken, decision points, summaries, basic information about who was present for what parts of a meeting and what people were attending, and the like. These processes might run on one machine, but would more likely take a whole set of machines, and they might provide quick results, or slow ones, after a delay. The company would have a range of annotations to choose from, depending on which end user applications they wished the Hub to support. Finally, the company would almost certainly already have a system in place for document management, which would ideally be accessible from the Hub, giving the system access to documents that are potentially relevant to a given meeting. If the recording set-up or video-conferencing system keeps track of which documents were accessed during which meetings, so much the better, since this is important for understanding what happened.

Once a company has installed their Hub, they can switch it on and the system will be ready to record meetings and produce annotations to accompany the signals that make up their meeting archive. For each meeting, when someone switches on the recording, as the same time as the audio and video signals went into the archive, the system would tacitly observe who was present and create a summary of the meeting, and note where to find the slides for any presentations given and documents that were consulted, and assuming the document management in order, where to find any minutes produced. It would also start up the components that produce annotations, which they would start putting into the Hub. The system would know when a meeting ended from when someone switched off the recording, or from a lack of activity in the room, and once the components had finished processing the meeting, if there were no other meeting for them to handle, they would go into standby until needed again.

At the same time as some components are producing annotations for a current or past meeting, end users will be running applications that pull this information out of the Hub and present it to them in one of a number of interfaces tailored to different tasks. There could be many users all running such applications at the same time to obtain different kinds of information, for data from any point between when the Hub was switched on until the moment just past. In addition, as time passes, new components might be brought on-line, applying new annotations, perhaps even to all of the existing data, in

service of new end user applications, and existing components that produce annotations might be replaced with better ones.

Although this vision for the Hub is far-reaching – such an infrastructure could not be easily be built today, but can be imagined for the near future – it informs the overall design for what we have implemented for use in the AMIDA project. The vision suggests our major requirement for its design: it must be a network resource that can handle access from many annotation producers and consumers at the same time, and be able to store, search, and retrieve very large amounts of data quickly.

3 The overall design

At the simplest level of description, our complete system for handling annotations brokers between the producers of annotations and their consumers. From the producer's point of view, it provides a mechanism for putting annotations into a permanent archive. From the consumer's point of view, it provides information about what types of annotations are in the archive and a mechanism for retrieving relevant annotations.

3.1 Producers

Annotation producers process one or more of the signals from a meeting recording. Typical annotations might include:

- words, as the output from automatic speech recognition;
- head orientation or direction of gaze;
- movement events, such as sitting, standing, or walking;
- decision points, saying a decision was made at a particular time;
- dialogue acts dividing words into e.g., questions versus statements.

Ordinarily, each component that produces annotation will produce one type. The annotations themselves typically take the form of a statement that from time X to time Y during the meeting, a particular thing was happening: Fred was standing, for instance, or Doris was speaking the word "control". Some types of annotations refer to the behaviour of an individual, and some to the behaviour of the group as a whole. A set of annotations of one type from one producer is an annotation stream.

It isn't enough for a producer just to start passing annotations in a stream into the Hub, since no component will be able to make use of those annotations unless they know what they are for, and how they compare to other, rival versions of the same information. For instance, there might be two speech recognizers hooked up to the same Hub and processing the same meetings: a fast one that yields poor results, and a slower one that gives better ones. Before producers start passing in annotations, they need to declare what type of annotations they produce, with what sorts of expected latency, so that the annotations can be used sensibly in end user applications.

3.2 Consumers

An annotation consumer can be anything that needs to read one or more types of annotations from the Hub. Often, a consumer will want all annotations of a given type for a given meeting. For instance, a meeting browser that shows transcription will want all words for every speaker in the meeting. Such a consumer effectively needs complete annotation streams. Sometimes a consumer needs a fragment with annotations for just part of the full meeting, and sometimes they may be interested in confluences of events that combine information from different annotation streams. Consumers can subscribe to data by query, where each query feeds its results to the consumer on a different socket. A query can match an entire annotation stream or filter an annotation stream to discard parts of it. The matching is done based on regular expressions over the individual properties for the annotation, such as the annotation type or the meeting which it describes.

For any given query, there is a basic expectation that the annotations will be fed to the consumer in temporal order. However, producers are not required to guarantee this ordering. It is the consumer's responsibility to check the ordering by testing incoming data against the timestamp of the last datum received. There are several processing metaphors that can be used to deal with producer data.

- If the consumer wishes to be fast but not guarantee results, it will simply buffer any data that is out of order, dealing with inserting it on a separate, lower priority thread. This is the normal metaphor for assistance during an ongoing meeting.

- If the consumer wishes to guarantee results are correct at a particular time, it will insert as it goes, making the processing less likely to keep up with incoming data.

Note that there is no reason why a producer should not be a consumer as well; for instance, a dialogue act recognizer will need access to output from the speech recognizer in order to operate. That is, producers are likely to operate on one or more of the signals from the meeting recording, but they may also take as inputs pre-existing interpretations of those signals, which are themselves expected to be found in the Hub.

3.3 Data and metadata model

In the data model employed by producers and consumers, there are two possible types of annotation streams. In the first, each element, representing one annotation, has an explicit start time, with an implicit end time which is the start of the next element. This gives a stream the semantics of a mutually exclusive and exhaustive sequence of intervals spanning the duration of a meeting. In the second, elements contain explicit start and end times. For either kind of stream, each element uses as its id something constructed using its start time and a unique identifier for the stream to which it belongs. The id of the stream uniquely identifies several properties of the annotation:

- the meeting with which the annotation is associated. This is necessary because there could be two simultaneous meetings in the Hub.
- the producer by which it was generated
- if the annotation is about an individual's behaviour, which individual it describes.

In addition to the start time and stream id, each annotation element has type information that gives its contents. The type information for each annotation element describes one or more attributes which can have numeric, enumerated or free-text type.

As well as the streams of annotations that are produced and consumed within the system, there are some kinds of information that just describe a meeting overall, not timespans within the meeting. Some typical kinds of metadata for a meeting are which group or purpose was involved; who was present; where the meeting was held; and what documents were consulted during the meeting. In addition, information declared by producers about data types and so on is also considered to be metadata. Although we could force all metadata into a fake kind of annotation stream, this would be rather inconvenient for developers of producers and consumers, since they would have to remember what conventions were used to do this. For this reason, we treat metadata using a separate metadata model.

3.4 Annotation archive (triple store) and Hub Base

Although producers and consumers use the typed data model that we have described, annotations are not actually stored in a database that retains the semantics implicit in this data model, because such a representation would require the underlying database schema to be changed every time a new kind of producer were connected to the system. This would ordinarily require the database to be taken down, which is undesirable for a system of this kind. There are many different possible ways of implementing an archive that is more robust in the face of change, although any successful implementation needs to be fast and able to store very large amounts of data. On the other hand, although it must be possible to put annotations into the archive, it is never necessary to remove them.. Our archive stores simple timed triples: object, attribute, value triples with an associated time stamp, leaving the task of translating between our typed data model and these triples to the middleware that intermediates between consumers, producers, and the annotation archive. For convenience, metadata is stored in the same annotation archive as the annotations, using set conventions about the time to use on the triple. Details about the triple representation used for both data and metadata are given in section 5.

The Hub Base is simply software that handles the triple store that comprises the annotation archive. When it receives a new triple from the middleware, it adds it to the archive. It also returns search results for queries posed by the middleware to the archive, using a query language based on regular expressions over the triple fields. In order to allow the annotation archive to reside on a different machine from producers and clients, its communication to the middleware is via sockets.

3.5 Middleware

In our architecture, the middleware is what bridges the gap between our data model and the timed triples of the annotation archive. The middleware consists of libraries that producer and consumer clients of Hub data can use to access the annotation archive indirectly. The libraries allow producers to express their annotations in terms of the typed data model, transforms those annotations into the underlying triple representation, and passes them to the Hub Base. Similarly, the libraries allow consumers to express queries in terms of the data model fields, transforms those queries into the simpler query language employed in the annotation archive, and passes those queries to the Hub Base for execution. In the middleware, the methods that producers and consumers use to add and access metadata conform to the metadata model and therefore are completely separate from those for adding and accessing data, even though underneath, both are stored in the same place.

In our architecture, it is technically possible for producers and consumers to bypass the middleware and work directly with the Hub Base. This is not recommended because it places an added burden on the producers and consumers; either they have to effect their own data transforms, or eschew semantics completely and hope that they underlying triple representation is as the developer thinks.

3.6 Running demonstrations and attaching non-Java producers

Although our overall concept for the Hub is sound in the way it covers the lifetime needs of a company or group, it does raise problems for development. Programmers developing new components or even just demonstrating end-user applications that deal with meetings as they happen need to be able to populate the Hub exactly as and when they need it. For this reason, we provide auxiliary software that can be used to create a new Hub and populate it, with the expectation that then it will be destroyed and a new Hub created for further testing. Since the AMIDA project has their "historical" annotations for selected past runs of producers stored in the format for the NITE XML Toolkit, this software takes that format, transforms it into a tab-delimited one where the fields match what is expected for the given annotation type, and stream that into the Hub. The software can be run as if in real-time, passing each annotation as its time comes. The final technique for streaming in tab-delimited data can also be used to connect non-Java producers to the Hub.

4 How to use the Hub and Middleware

The architecture for the Hub/Middleware implementation is shown in Figure 2. The client library contains a Hub client plus the middleware to transform Producers' data elements into Hub elements and Consumer queries into Hub queries, amalgamating the results and passing back as expected.

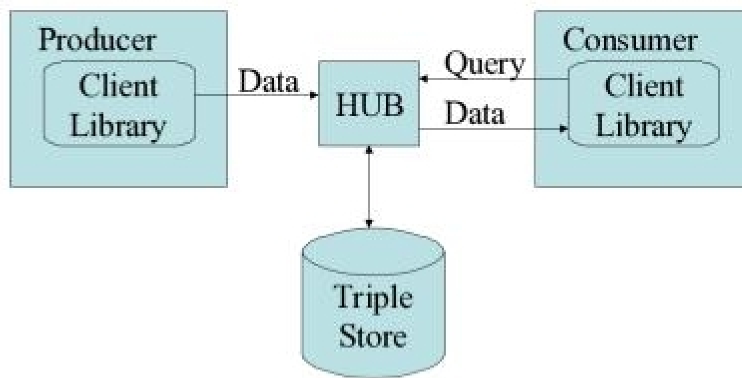

Figure 1 Hub middleware architecture

To set up your own Hub, you first need to start a MySQL database, load it with the provided database schema, and then start the Hub process that connects to it. Instructions for doing this along with code examples that produce and consume data and metadata are provided.

4.1 Connecting to The Hub

Connecting to the Hub must be done by producers and consumers alike. It is done using a simple Java call:

```
Midpoint midpoint = new HubMidpoint();
```

The location of the Hub is made known to Java using a set of *Java properties*:

Property name	Default value	Description
ch.idiap.hub.debug	false	command line debug info
ch.idiap.hub.timeorderedoutput	false	if true, all query output is guaranteed time-ordered
ch.idiap.hub.host	localhost	the name of the machine runing the Hub server
ch.idiap.hub.table	test	name of the DB table
ch.idiap.hub.jdbc	jdbc:mysql://localhost/test	name of DB, protocol and optionally username and password
ch.idiap.hub.port	4000	the port over which the Hub connection is made

4.2 Example Consumer

Consumers of meeting data wish to ask questions bounded by meetings and sets of meetings, participants and groups, or particular time periods within or between meetings. The middleware privileges this metadata as distinct from annotation data.

Annotation data from producers is treated as essentially uniform. As described below, producers are only allowed to publish streams of *DataElements*, each element having a pre-published type declaration. Each element must be associated with its type value(s) and its producer, and may also be associated with a meeting, participant, start time, end time and confidence score. Those properties can all form part of a consumer query.

Consumers query the middleware by filling in the slots of a Query element and submitting it. Matching data elements are returned one-by-one using callbacks to a result handler. For example:

```
Query q = new Query("word");        // looking for elements called 'word'
q.setMeeting("ES2008(a|b|c)");     // in meeting ES2008a, b or c
middleware.performQuery(q, this);
```

The second argument to *performQuery* is a *ResultHandler* where the callbacks containing query results are sent.

4.3 Example Producer

Producers can register one or more *ElementClass* that define the *DataElement*s they produce. Each has an element name and a type, and may have an arbitrary number of further attributes, each of which has its own type. The types may be enumerated, free text, or numeric.

Here's some code for a producer that will output ASR with an estimated word-accuracy rate of 40%, 5000ms after the end of the speech element arrives:

```
Producer asrproducer = new Producer();
asrproducer.addElementProduced(new ElementClass("word",
                 new ElementType(ElementType.FREE TEXT)));
asrproducer.setEstimatedAccuracyMillis(5000);
asrproducer.setEstimatedAccuracyPercentage(40);
asrproducer.setOutputType(Producer.OUTPUTS AT END);
midpoint.registerProducer(asrproducer);
```

4.4 Example Metadata query

Metadata requests are serviced by middleware in a different manner to consumer queries. Instead of sending individual results to a handler, metadata requests return a complete set of results up to the current time. As an example, this code requests meeting information from the archive for a meeting named *ES2008d*, then requests its list of participants for that meeting. This code assumes a Hub connection as shown in 4.1.

```
Meeting meeting = midpoint.getMetadata().getMeetingFromID("ES2008d");
List participants = meeting.getParticipants();
```

The list of participants is correct at the time of query, but if the meeting is ongoing it is possible that the participant list will not be the same the next time it's requested.

4.5 Package Structure

The structure of the packages of the middleware implementation is shown in Figure 2:

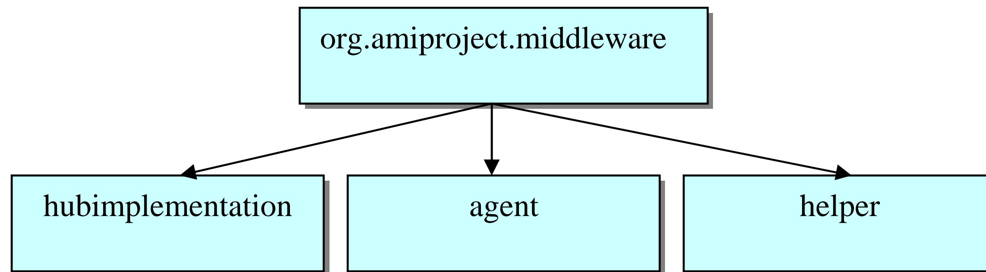

Figure 2 Hub middleware package structure

The *org.amiproject.middleware* package contains a set of interfaces defining how client programs interface to the Hub. The *hubimplementation* sub-package is our implementation using the Hub. The *agent* package contains example producer and consumer agents on which users can be base their own producer and consumer code. Finally, The *helper* package provides some abstract classes useful for developers.

5 Underlying Triple Representation

Though it is not necessary for end users to know how data is actually stored in the Hub, it is described here for completeness.

5.1 Data Elements

All data will be stored in the Hub as timed triples with IDs of this form:

<meeting-id>.<element-type>.<speaker-id>.<producer-id>.<numeric-id>

Time is expressed universally in milliseconds since the beginning of the 1970, as is traditional on Unix systems. The speaker is optional because some elements like topics are not associated with any speaker. If a producer has declared an element with more than typed attributes it will be represented by a set of triples with the same ID:

Time	ID	Attribute	Value
88886666	ES2008d.link..p3.98	starts	http://a.b.c/agenda.ppt

88886666	ES2008b.link...p3.98	rank	1

Notice that the ‘starts’ triple is overloaded with the value of the first attribute declared on the type. It can be retrieved via its original attribute name (if declared) or as the type-value of the element as a whole.

Middleware ensures a value for every declared attribute is stored for every data point (even if the producer fails to fill some slots). That allows the query-answering module to know exactly how many triples to expect before passing on data elements to result handlers.

5.2 Metadata Elements

This table describes how metadata elemnts are stored in the Hub as of version 1.3 of the middleware. Since metadata is untimed, we use set conventions for the time to associate with each type of metadata. For metadata that describes a meeting, the meeting start time is used. For metadata that describes a participant, document, or location, we used the time that the information is added to the annotation archive. For metadata that describes a producer or consumer, one set of triples is placed at the time that the component registers with the system and another at the time when it unregisters.

Time	ID	Attribute	Value
88880000	meeting.ES2008d	starts	-
88990000	meeting.ES2008d	ends	-
88990000	participant.FEO008	native-language	German
88990000	participant.FEO008	english-experience	2 years
88880000	meeting.ES2008d	participant1	FEO008
88880000	meeting.ES2008d	participant2	MEE010
88880000	meeting.ES2008d	location	EdMeetingRoom
88880000	role.ES2008d.role1	FEO008	PM
88880000	role.ES2008d.role2	MEE010	ME
88880000	location.ES2008d.loc1	MEE010	Home_MEE010
88880000	document.doc1	participant	FEO008
88880000	document.doc1	filename	http://x.y.z/presentation_kick_off.Rose.ppt
88880000	document.doc1	meeting	ES2008a
88880000	document.doc1	type	presentation
88880000	document.doc2	filename	http://x.y.z/ES2008d.Closeup1.avi
88880000	document.doc2	meeting	ES2008d
88880000	document.doc2	type	media

88880000	document.doc2	participant	MEE010
88889999	producer.p1	registers	
88889999	producer.p1	output_time	OUTPUT_AT_END
88889999	producer.p1	estimated_lag	11111
88889999	producer.p1	estimated_accuracy	45
88889999	producer.p1	produces_element	word
88889999	producer.p1.word	element_type	type1
88889999	producer.p1	description	ASR producer for AMI meetings
88899999	producer.p1	unregisters	
0	elementtype.type1	type	STRING
0	elementtype.type2	type	ENUMERATED
0	elementtype.type2	value1	deictic
0	elementtype.type2	value2	discursive
0	elementtype.type3	type	NUMBER
88880000	consumer.c1	registers	
88880000	consumer.c1	description	content linking display
88889999	consumer.c1	unregisters	

Media files are treated as a special kind of document. Documents can currently be of type unknown_type; presentation; agenda; minutes; figures (spreadsheet); summary; meeting_extract; media.

6 Discussion

As described, our overall system for handling annotations suffices for creating initial prototypes for AMIDA's main target applications, *engagement support for remote meetings* and *meeting information access*. However, there are a number of issues with this design which future end user technologies may find too limiting, and which may require embellishment of the current design. We discuss the most important of these here.

The first is that sometimes it is useful to know not just about annotations of a particular type or set of types, but also how annotations of two or more types relate to each other temporally. For instance, one might wish to know about facial expressions that occur just after decision points, or about pointing incidents and uses of deictic pronouns such as "this" in close temporal proximity. At the moment, our system allows consumers to subscribe to the annotation streams involved and filter a single annotation stream base on properties of the elements it contains, but leaves the consumer to its own devices when it comes to evaluating conditions across different annotation streams. This could in theory result in a great deal of data being passed to a consumer just for it to be thrown away. It

would be possible to add some more complex query operators that prevent this difficulty. The most useful operators would probably test for element pairs of mixed type that occur within a given time window plus the same test with an ordering specified for the types.

The second is the fact that the current data model only covers annotation streams where each element has a start and end time, whereas some kinds of annotation are effectively untimed. For instance, consider the case of an abstractive summary. Such a summary is an untimed but ordered set of sentences. It is possible to cram untimed data into our stream model by assigning fictional timings to the elements representing (in this case) the sentences, using some convention such as placing the first at the start time of the meeting and the rest at regular spaced intervals, with a flag at the end indicating that the data is complete. This has the advantage that it requires no changes to the Hub, but violates the principle of our middleware that the semantics for the data should be clear. Thus, we may wish to add more apparatus to the middleware that grounds out to this representation at the lower level, but treats untimed data distinctly from the point of view of producers and consumers.

Third, in previous work on the AMI project, some annotations have been related to each other in ways that are not temporal. For instance, each sentence from an abstractive summary has been associated with one or more dialogue acts drawn from across the streams of dialogue acts corresponding to each of the meeting participants. This allows the creation of end user tools in which the user can read a summary and then navigate to the most relevant parts of the meeting record. Similarly, pairs of dialogue acts can in theory be related to each other, for instance, as question and answer, but it is difficult to put a temporal bound on how far this relationship might stretch. If such relationships become important to the technologies we build, some facility for them will have to be added to the system, but with the proviso that this facility must above all be efficient in use. One possible representation would involve placing information in a separate stream expressing the relationship redundantly at the start time of both annotations involved, so that it can be found efficiently.

Fourth, at present the annotation system does not enforce that consumers and producers must access the annotation archive via the middleware, but if it did, we would be able to change the underlying data representation without adversely affecting end user technologies. Our current arrangement has the potential to be brittle. As the archive grows, we may need to think about investing in more clever ways of storing the base data. Thus, we should consider making use of the middleware mandatory.

Finally, in systems with a great many producers for annotations of different types, consumers may need a better way of sorting through what is available to them. Although the middleware imposes a semantics to the Hub's simplistic triple store, consumers still need to know what the various types produced will be; which one contains transcribed words, for instance. That is, the system assumes a common vocabulary for producers and consumers that may not always be feasible in practice. One way to provide support for vocabulary differences would be to include an ontology of types to which both consumers and producers could refer as a translation aid, without imposing the hard

constraint that the actual type names be used in the data they pass. This ontology would be most helpful to developers of end user technologies if it were human-readable.