Modeling, obtaining and storing data from social media tools with Artefact-Actor-Networks

Wolfgang Reinhardt, Tobias Varlemann, Matthias Moi and **Adrian Wilke**
University of Paderborn
33102 Paderborn, Germany
{wolle,tobiashv,moisun,wilke}@uni-paderborn.de

Abstract

Social interaction between people has peerlessly changed with the availability of the Internet and the World Wide Web. The Internet brought new ways of communication technologies to live and enhanced people's reachability, augmented possibilities for personal presence and the sharing of information objects. People are engaging in social networks in a steadily growing manner and share information objects within their communities. The high initial amount of data in such networks can serve as foundation of serious investigations towards social interactions of communities of learners. In this paper we introduce the technological foundation and architecture to model, obtain and store such user and object information in so-called Artefact-Actor-Networks. Artefact-Actor-Networks combine classical social networks with artefact networks that are constructed by the use of the information objects and their connections.

1 Introduction

The Internet has evolved to be the most frequently used medium of the 21st century and it is steadily growing. The so-called Web 2.0 movement [O'Reilly, 2005] engaged people in the active production of content on the Web and fostered technology-mediated human interactions in social networks. People are heavily taking part in social networks for individual learning and knowledge work as well as for leisure activities. The huge amount of data in such networks can serve as foundation of serious investigations towards social interactions in communities of learners, knowledge workers or people spending their leisure activities online.

There already is an existing body of knowledge about the properties of social networks such as Facebook (see [Ellison *et al.*, 2007]), Twitter (see [Java *et al.*, 2007; Ebner and Schiefner, 2008; Reinhardt *et al.*, 2009a]), in Social Bookmarking systems and learning object repositories [Vuorikari, 2009] but most often this work stay focused on the relational and structural part of social network analysis [Granovetter, 1983]. Nevertheless, those studies are often combined with qualitative data from interviews or online questionnaires and thus can serve as useful information source for future research. Despite their undoubted usefulness, they do often not consider the artefacts resulting from online interactions, their content, structure and relations. Furthermore, the relations between artefacts and their creators, editors or linkers are not considered for making claims about a community. In [Reinhardt *et al.*, 2009b] we introduced the model of Artefact-Actor-Networks (AANs)[1] which tries to overcome those limitations by considering both content of artefacts and the relations to actors interacting with them. As artefact we consider any digital information object that is shared within a communitiy (examples are simple HTML pages, status updates in microblogging services, entries in blogs, social bookmarks, online available documents, photos in picture sharing services and many more).

In this paper we focus on the technical side of AANs and introduce the architecture to model, obtain and store such user and object information.

2 Obtaining and modeling data from social networks

In this section we describe the concept of Artefact-Actor-Networks and the ontologies in place to model both actors in social networks and the artefacts resulting from their interaction. Moreover, we will reference existing standards and vocabularies for modeling objects and their interactions in online communities as well as metadata describing the respective properties of such objects.

2.1 Artefact-Actor-Networks

Artefact-Actor-Networks (AANs) are an approach to semantically intertwine social networks with so-called artefact networks. The theoretical model and a first implementation were first introduced by [Reinhardt *et al.*, 2009b].

To connect artefacts and actors with each other, semantic relations are required. Relations in the network are connecting objects by a semantic context like *isAuthor* or *isRightHolder*. With the help of Artefact-Actor-Networks participation in the life cycle of artefacts as well as significant connections to involved actors will be outlined. Artefact-Actor-Networks consolidate multilayered social networks and artefact networks in an integrated network. Therefore, we consider the communication and collaboration with each communication tool or artefact supply (e.g. Twitter, chats, e-mail or scientific documents) as a single layer of the respective network. We unite these single layers in both social and artefact networks to consolidated networks that contain all actors and artefacts respectively (cf. figure 1). While in the consolidated social network we can only make statements concerning the relations between actors and in the consolidated artefact network we

[1] See `http://artefact-actor-networks.net`

can only analyze the relations between artefacts, Artefact-Actor-Networks (cf. figure 2) also contain semantic relations between actors and artefacts.

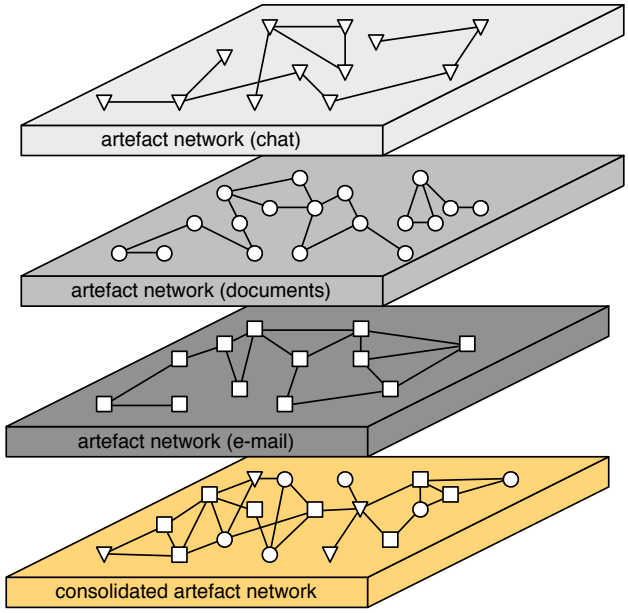

Figure 1: Consolidated artefact network resulting from three layers

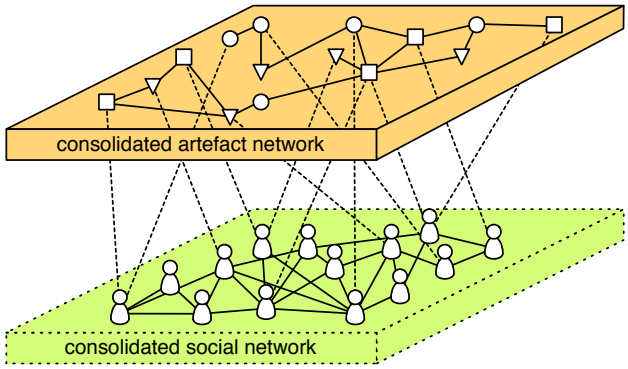

Figure 2: Artefact-Actor-Network with semantic relations between artefacts and actors

In Artefact-Actor-Networks we discern three types of semantic relations: those between artefacts (ART^2 relations), those between actors (ACT^2 relations) and finally relations that exist between actors and artefacts (AA relations). Each kind of relation can be used for certain types of analyses and supports a different type of awareness in cooperative settings. ACT^2 relations describe the nature of relationships between involved people. They characterize simple connections, friendships or kinships. Furthermore, they can show the kind of media people are communicating with. The Friend of a Friend (FOAF) project [FOAF, 2010] developed a RDF vocabulary to express interests, connections and activities of people. ART^2 relations on the other hand provide information on how artefacts are connected. The Dublin Core metadata standard and the SIOC project currently provide an expedient starting point [DCMI, 2010; SIOC, 2010]. Lastly, AA relations describe the semantics of relations between actors and the artefacts they interacted with. Dublin Core and SIOC provide useful relations to build upon, but the learning objects metadata standard (LOM) could be taken into account as well.

2.2 Relevant ontologies

During the modeling of the application domain however, we found out that we needed to extend the beforementioned ontologies[2] and vocabularies in order to cover the specifics of interaction with social media in learning networks. Thus, we created several ontologies for the social media services we are analyzing (see Section 3.3) and made our ontologies publicly available[3]. The relations build on already existing standards for the modeling and storage of metadata and are further extended by our application. Figure 3 shows a simplified overview of the ontologies used in Artefact-Actor-Networks, where AANBase defines the basic entities Actor, Artefact and Keyword. AANMeta is an ontology that allows the aggregation of multiple actors (online handles) in one real person and the relationship between so-called groups to real people, their actors and artefacts related to a group (for example artefacts that are tagged with one of the groups tags). The AANOnline ontology is used to differentiate between artefacts resulting from online actions and such artefacts that relate to activities taking place offline. The more specific tools and the respective ontologies are located towards the right of Figure 3.

3 The architecture of Artefact-Actor-Networks

The architecture of Artefact-Actor-Networks is composed of a backend to which several frontends can be connected. The backend is responsible for processing and storing data and exposing this data to the frontends by well-defined interfaces.

The architecture is based on the OSGi Service Platform, which is a component framework based on the Java platform. Due to the use of a specified component structure the backend is easily expandable and components can be deployed during runtime. The communication between the components is ensured with techniques from OSGi.

It is the backend's task to examine contents, to analyze, store and provide the annotated data. These four tasks are reflected in the according blocks in the AAN architecture (cf. Figure 4).

The crawling block loads contents and generates (according to the ontology) structured data. The datastore block stores all data generated and serves as the connection between all the existing blocks. The analyzer block contains several components with which a further refinement of the generated data is achieved. The fourth block provides the interfaces for various frontends. These blocks and the contained components are described in the following.

[2]Ontologies are a common way to model problem domains in an extensible and open format that is usable in various contexts. [Lohmann and Riechert, 2010] note the very precise and popular definition of the term ontology given by [Gruber, 1993] who notes that an ontology is "*a specification of conceptualization*" . Furthermore [Maalej *et al.*, 2008] prod to the fact that ontologies normally are valid for a much longer time than conceptual models for example, as they describe a broader application domain and some more general knowledge facts.

[3]See `http://artefact-actor-networks.net/ontologies/2010/03/`

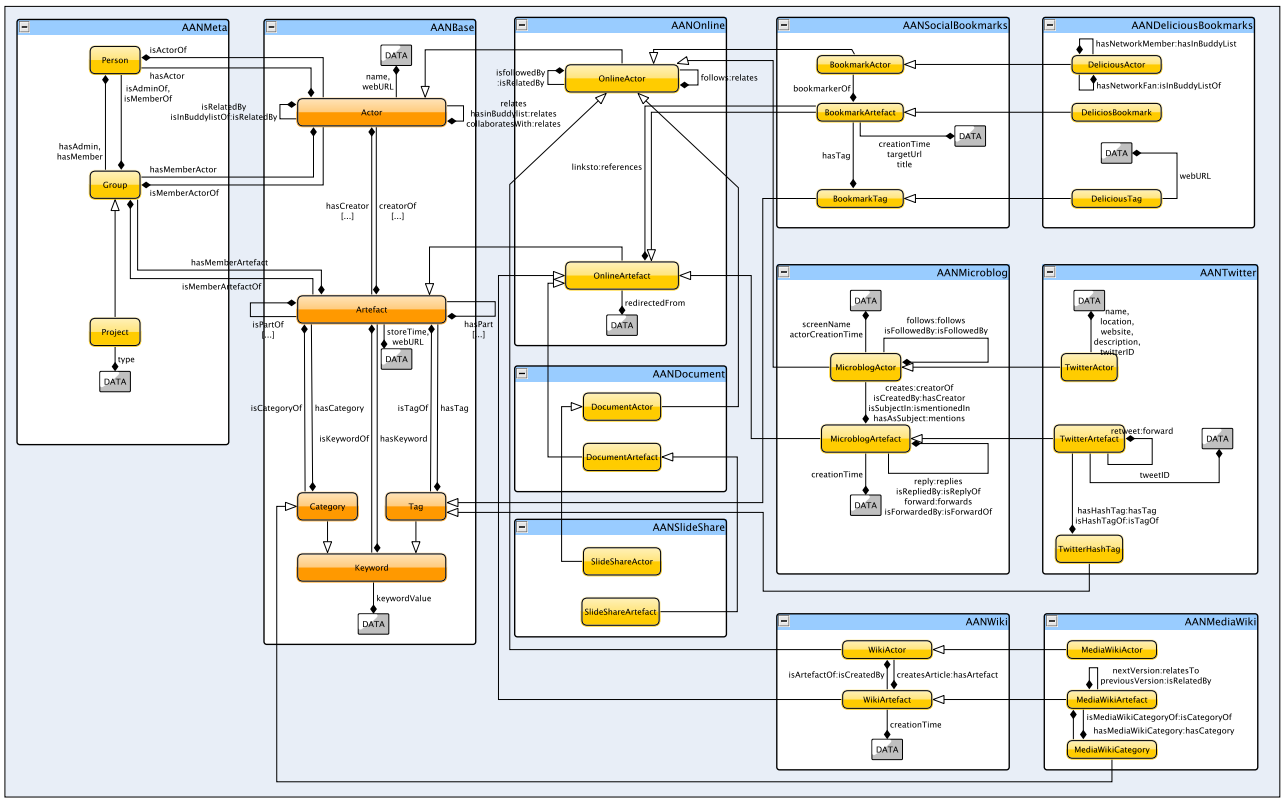

Figure 3: Simplified overview of the AAN Ontologies

3.1 Crawling-Block

At the crawling block, requests for content analysis are processed. These requests are passed to the crawling block via OSGi-Services and analyzed by a processing chain. There are two interfaces: The components Crawler and CrawlerManager. The CrawlerManager contains high-level functions by which the Crawler is controlled. The processing chain is controlled by the crawling component. It consists of the components Accessor, which is reading the content, MimeTyper, to identifier the type, and Parser, which analyzes the content.

Crawler

The Crawler component provides low-level functions to process tasks. Tasks are committed as URIs of the content. For such an URI, the Crawler executes the processing chain by loading, determining the MIME type and analyzing the content. The selection of the Accessor component depends on the used protocol and the URI. Accordingly, the MIME type is determined, what in turn selects the Parser to store contents in the datastore block. While the Parser is chosen, it is taken into consideration, if a specialized Parser exists that is able to handle variations of the detected MIME type. An example is a request of a web service, which supplies a MIME type *text/xml*. The use of a Parser that is able to handle this special type is more practicable than the use of a Parser, which can process all XML formats.

The processing of tasks occurs asynchronously by the use of a thread pool with which several tasks can be executed parallel.

CrawlerManager

A superordinate of the Crawler component is the CrawlerManager, which is using the services of the Crawler. The CrawlerManager provides functions of higher levels than the Crawler. Whereas the Crawler is receiving tasks via an exact URI, the CrawlerManager is designed to handle more complex jobs. It is possible to deal with individualities of web pages or investigate the structure of a page by following hyperlinks in HTML documents.

To handle special tasks, there can be various implementations of the CrawlerManager. Two examples, the GenericCrawlerManager and the MediaWikiCrawlerManager, can be seen in the architecture (figure 4). The GenericCrawlerManager is able to process timed tasks, which can generate follow-up tasks. The MediaWikiCrawlerManager is specialized to the structure of MediaWiki pages and is able to crawl contents of an entire wiki.

Accessor

The first element of the processing chain is the Accessor component. It is responsible for reading content. The component is responding to the Crawler, which hands over an URI of the content to load. The Accessor component accesses the resource, stores it in a local file, and returns a reference to the file.

Different types of the Accessor component are conceivable. Those could enable the access to protocols like HTTP, FTP, SMTP or SVN.

MimeTyper

For further processing of a resource by a Parser the MIME type of resource is determined. This improves the choice of an appropriate Parser in the last stage of the processing chain. The MimeTyper is loading the temporary, local file of the content to analyze. Then it determines one or more MIME types of the content and returns it to the Crawler.

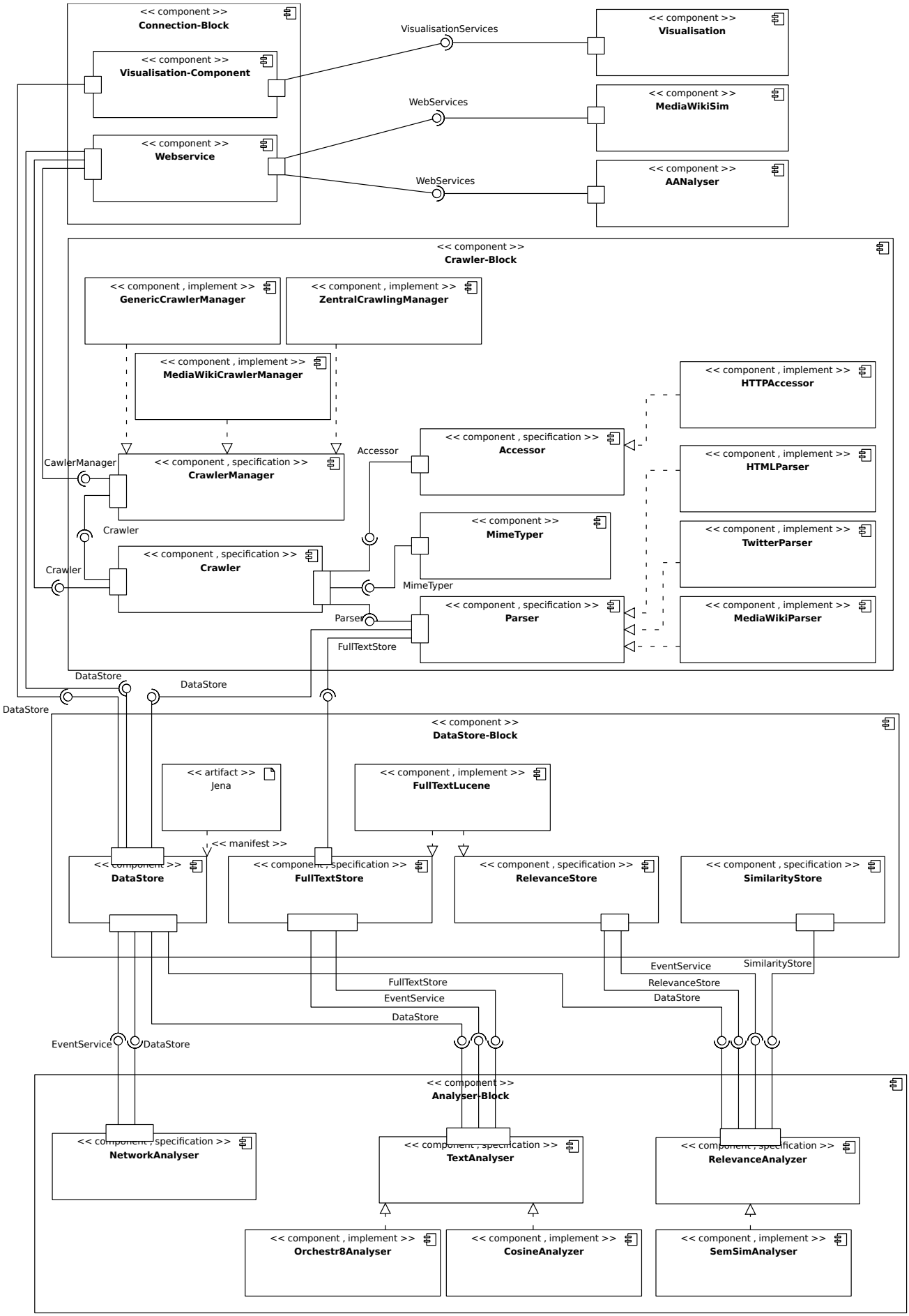

Figure 4: The AAN architecture

Parser
The Parser is the last stage of the processing chain. It extracts relevant information from the resources loaded by the Accessor, to build up an Artefact-Actor-Network. Different variations of Parsers are distinguished by the Parser type and the MIME types which can be processed. Types are setting in advance the order, in which the Parsers are chosen. Parsers which can handle a specialized MIME type are preferred to general Parsers. This guarantees the choice of a Parser which can process contents optimally.

3.2 DataStore block

The DataStore block is responsible for the storage of all data produced in the application. For this purpose, four components are available to accommodate the differently structured data. Other components will be notified of any changes via an event service. All components in the DataStore block send custom events for communication purposes.

DataStore
The Parsers generate RDF data from the analyzed contents that is stored in the DataStore according to the respective ontology. RDF data provides several advantages for Artefact-Actor-Networks. In addition to the flexibility of the data structure and the ability to generate descriptions of the data at runtime, there is a wide range of storage engines that are adapted to the processing of RDF data. The backend uses the Jena Framework[4] to store and query the data. With Jena data can be queried both in source code and via the RDF data query language SPARQL[5]. The DataStore stores the data in a file format defined by the Jena Framework but one can also use in-memory storage or a RDBMS.

FullTextStore
Some analytical techniques for text-based content – such as keywords, named entities or related languages – require the existence of a full text. The texts can be extracted by a Parser and will be stored in the FullTextStore. The FullTextStore will notify components in the analysis block about the existence of a full text that can be retrieved via an event system.

RelevanceStore
The RelevanceStore holds information about the relevance of a keyword. This relevances are used during the calculation of semantic similarity of artefacts. In our architecture the FullTextStore and the RelevanceStore were incorporated in one component. We use the Apache Lucene engine to efficiently store the full texts and to query the FullTextStore for the relevance of single keywords.

SimilarityStore
The SimilarityStore manages all calculated similarity values between artefacts, actors, persons and groups. Similarity values are created by various similarity analyzers, which store their results in the SimilarityStore according to an analyzer-specific order.

3.3 Analysis block

The Analysis block includes all components for analyzing stored data of components from the DataStore block. Components in this block only listen to events fired from DataStore components. We distinguish between NetworkAnalysers, TextAnalysers and RelevanceAnalysers. NetworkAnalysers listen to events fired from the DataStore, Textanalysers listen to events from the FulltextStore and RelevanceAnalysers to events from the RelevanceStore.

NetworkAnalyser
NetworkAnalysers are using the data pool of the DataStore to analyse network structure. They will be informed to every change in the data pool of the DataStore.

TextAnalyser
TextAnalysers start working upon the reception of FullTextStore events. Currently we implemented Orchestr8Analyser, OpenCalaisAnalyser and CosineAnalyser. The Orchestr8Analyser and OpenCalaisAnalyser are making use of the webservices of Orchestr8 and OpenCalais respectively to extract keywords and named entities from a given text. Extracted data will be stored as additional parts of resources.

RelevanceAnalyser
RelevanceAnalysers are components to determine the similarity of resources. An example for such an algorithm could be the SemSim algorithm that we developed (see [Reinhardt *et al.*, 2009b] for more detail on the algorithm). SemSim allows the calculation of semantical similarity based on the keywords and named entities stored for artefacts. The calculated values are then stored in the SimilarityStore.

3.4 Connection block

The fourth block in our architecture is the Connection block that encapsulates Webservices and the visualization component. Both components expose data stored in Artefact-Actor-Networks or API functionalities to external consumers over HTTP interfaces.

Webservices
Internal server components such as the CrawlerManager, the CrawlerManager or the DataStore are providing their functionalities via webservice interfaces as an API to external applications. Using the API, crawling jobs can be placed or data from the DataStore can be requested. The DataStore provides tailored API functions as well as a SPARQL interface to the AAN.

Visualization component
The visualization component converts the contents of Artefact-Actor-Networks into a XML-based graphics format that can be displayed and explored in the appropriate viewers. The preparation of those files is costly and will be a great burden for the AAN server. In order to reduce the efforts we implement a caching strategy that will only keep those projects up-to-date that are being accessed regularly.

3.5 Technical note

The architecture is based on the OSGi service platform[6] a component framework based on the Java platform. The OSGi service platform is a specification of the OSGi Alliance, which has been implemented in various implementations. Amongst those implementations are Eclipse Equinox[7] and Apache Felix[8]. The frameworks support the live deployment of components, which allows to add a new Parser without the need to restart the server.

[4] http://openjena.org/
[5] http://www.w3.org/TR/rdf-sparql-query/
[6] http://www.osgi.org
[7] http://www.eclipse.org/equinox
[8] http://felix.apache.org

OSGi also allows the dynamic communication of components with each other. For this purpose, a service system is used that allows components to register services that then can be used by other components. The AAN architecture makes heavy use of this approach, as all components provide specialized services and await use. As an example, each Parser has to provide a service that can be accessed via the two methods *isParsable* and *parse* of the Crawler component.

4 Social media tools under investigation

In its current implementation of AANs we store and analyze data from four different social media tools: (1) Twitter, (2) Delicious, (3) SlideShare and (4) MediaWiki. All of the tools are used by researchers during their daily work routines (see for example [Heinze *et al.*, 2010] for an inspection of tools used by researchers) and make specific demands on the respective components in the AAN architecture. In the following we present the specifics of the single components.

4.1 The specifics of the Twitter component

Twitter[9] is a microblogging service, which allow users to publish short messages with an length under 140 characters. These messages are typically public and can be viewed over different channels. The *TwitterParser* component use the answers of the TwitterAPI in XML or JSON format to parse the information of a single Tweet(Status), a Twitteruser, a users timeline, the followers of a User or a search request. The *TwitterParser* possesses a special component for each of this functions which will be described now.

The *StatusComponent* parses the XML answer of the TwitterAPI */show/status* request. It extracts the information about the status and the user who created it. The extracted information is kept in the *DataStore* as listed below:

CreationTime the creation time of the status.

Statusid the Twitter id of the status.

Replyid the Twitter id of the status to which the current status is the reply.

User the information of the creator of this status. This will use the *UserComponent*.

WebURL the URL to this status as an HTML page.

Hashtags the hashtags of this status.

ExternalLinks hyperlinks that this status may contain.

Text the full text of the status that will be stored in the *FullTextStore*.

TwitterAPI */show/user* requests are processed in the *UserComponent* which extracts information about the Twitter user and stores them in the *DataStore*. The extracted information are listed below:

UserID the Twitter id of the user.

Screenname the screen name of the user.

Username the real name of the user.

Location the location where the user lives.

Description the description of the user which was entered in Twitter.

URL the internet address of the user.

CreationTime the date at which the user registered at Twitter.

Last Status the last status of the user (will passed to the *StatusComponent* for analysis).

The TwitterAPI response for a timeline contains a series of statuses. This series will be separated by the *TimelineComponent* into the single statuses. At the last step the *TimelineComponent* will forward each single status to the *StatusComponent* which will extract the information.

The */show/followersid* API call responds with a list of TwitteruserIDs which will be parsed and returned to the *Crawler* as follow-up links, which can be followed by a *CrawlerManager* to parse the user information of the followers.

The response from the TwitterSearchAPI is computed the same way as followers. The StatusID will be extracted from the response and returned to the *Crawler* as follow-up links. This is necessary because Twitter uses a different data structure in the SearchAPI as in the rest of the TwitterAPI.

4.2 The specifics of the Delicious component

One of the integrated data sources is the social bookmarking service Delicious[10]. Delicious can be used to store personal bookmarks on the web and share them with others. During the creation of bookmarks users have the opportunity to add notes and tags to describe and categorize their input. By adding this additional data, especially the tags, artefact-networks are created. On the one hand bookmarks of different users form networks by relations resulting from their tags. On the other hand all bookmarks of a user are connected to the user himself. Besides these artefact-networks, actor-networks can also be found at Delicious. These are formed while users add other Delicious users to their personal network. By these relations, some users are connected indirectly, as well as their bookmarks are connected additionally. In summary, Delicious provides both types of networks that form the base of Artefact-Actor-Networks. What is the most practicable way of proceeding to get it?

Data access

Delicious offers a huge amount of possibilities for developers to access the available data[11]. Depending on the desired outcome, one can choose from different interfaces, e.g. an API, feeds or link-rolls. Generally, one of the most applied ways to access data is by the use of an API. We also tested this way for applicability to our system. The offered API-methods are custom-made for the access and use of a users personal data. A user can create, edit and receive personal bookmarks, tags and tag bundles. As the idea of AAN is the extraction and analysis of public data, and the need of a user-authentication by the API is a hindrance, there was a need for more useful data access.

A more utilizable approach to get data is the use of Delicious feeds[12]. Feeds are offered in JSON and RSS format and provide an access to public data. It is possible to get the latest bookmarks, tags and network members of a specified user. Furthermore, requests for bookmarks can be refined by combining a specific username and tags. Moreover, recent bookmarks for an URL can also be accessed what forms an extensive base for information retrieval.

[9] http://www.twitter.com/

[10] http://delicious.com/

[11] http://delicious.com/help/tools

[12] http://delicious.com/help/feeds

As feeds are mainly used for receiving the latest information, most of the feeds are provided as a list of recent bookmarks. Furthermore, the Delicious feeds are limited to 100 entries per request and as there is also a limitation of one request per second we encountered a restriction for crawling the entire bookmarks of a user. This fact was partly solved by multiple recursive requests. If a user has described his bookmarks with the tags A, B and C, first the bookmarks described with tag A are requested. If the returned set of bookmarks amounts to 100 entires, a combined request of tag A and B is sent. If the result of this request amounts to 100 entries again, a refinement by an additional tag is used. Otherwise, the bookmarks of the tags A and C are requested to get a result that is as complete as possible.

Within our system, the *DeliciousParser* is setting properties for contents of specific feed calls. With these properties, extracted tags and the count of returned artefact entries are stored. This forms the basis, by which the *DeliciousCrawlerManager* is generating feed URLs to follow up crawling a complete set of bookmarks.

Finally, the received feed-data is mapped to the defined ontology and added to the AAN model.

4.3 The specifics of the SlideShare component

SlideShare[13] is a Web 2.0 platform which offers users the possibility to share presentations and documents. A user can upload files in PDF and common office formats and is able to define metadata like tags, category and visibility information. Published slides can be favored, rated, commented on, downloaded and shared with others.

For developers, SlideShare provides an API[14] with which public and private data can be accessed. Public API methods require an optional user authorization. By using public methods, developers can request documents related to users or tags. Additionally, a user's tags or contacts can be requested as well.

For accessing data of the SlideShare network, the expandability of the AAN framework is used. Here we took advantage of the clear URL scheme of the API methods and thus the existing components CrawlerManager and ZentralCrawlerManager work together with the SlideShareParser, a specialized parser to analyze the incoming SlideShare data. New crawling tasks are added to one of the CrawlerManagers. If such a task consists of a SlideShare URI, the specialized SlideShareParser determines that it is able to handle the given input. If the parser is chosen it firstly analyzes the URI scheme. In the following, artefacts, actors and metadata are extracted and stored accordingly to the specified ontology. Finally, API URLs of related actors, artefacts and keywords are generated and added to the crawling queue.

4.4 The specifics of the MediaWiki component

The MediaWiki component was designed to receive as much information as possible from a MediaWiki installations such as WikiPedia. First we designed a specialized MediaWikiCrawlerManager, which is able to control the crawling process to crawl and observe a complete MediaWiki or just a single page. Furthermore, we implemented the MediaWikiParser, who's task it is to parse input that was received by the Crawler (see Section 3.1). The MediaWikiParser stores extracted information like internal and external links or information about the author within the DataStore component. The full text will be stored with the FullTextStore component. We distinguish between three different types of jobs handled by the MediaWikiCrawlerManager, which will be discussed below.

Crawling single pages

In this case, the MediaWikiCrawlerManager handles the job in four steps. First a unique URL as a permalink[15] will be generated. As discussed before every object, like a MediaWiki article is represented as a unique artefact in the Artefact-Actor-Network. Secondly the MediaWikiCrawlerManager generates an MediaWiki API-Query of the type 'parse'. The required information is initially received from the MediaWiki server of the article. Note, that this the query is not executed by the MediaWikiCrawlerManager but only the appropriate URL will be created. In the third step the MediaWikiCrawlerManager calls the Crawler to add a new crawl task. The resource will be fetched by one of the accessor components. If the MediaWikiParser is registered and started, the MediaWikiParser parses the resource because of the distinction between special and general parsers.

Further Properties By adding a new job to crawl only a single page one is able to define the properties to resolve internal and external links by specifying a depth value. If a the depth for internal link is 1 for example, the MediaWikiCrawlerManager will add new jobs for all received internal links.

Crawling a full MediaWiki

The MediaWikiCrawler can crawl a complete MediaWiki with all its articles in the latest revision or with all its articles with a specifiable number of revisions. By adding a job to crawl a complete MediaWiki, the MediaWikiCrawlerManager executes an API query to receive a list of all articles, including the latest revision information about an article. If one wants to get more then the current revision of an article, the MediaWikiCrawlerManager executes queries to get basically needed information about each revision. With this information, the MediaWikiCrawlerManager finally generates MediaWiki API queries of the type parse to add new single page jobs. Each of the generated jobs will be handled like described in the section about crawling single pages. Another important note is the limitation on the MediaWiki API. Only 500 article or revision entries can be received with one query. This is solved by executing serial queries.

Further properties For this job type the properties revision count and the depth of external links can be specified. The property revision count can be a positive integer values or -1 to crawl all revisions of all pages. The parameter about the depth of the external links is the same as described in the section about crawling single pages. It will be propagated to each created single page job.

Observing a MediaWiki

To keep AAN data up to date, the MediaWikiCrawlerManager supports the observation of a MediaWiki. After crawling a full MediaWiki initially, it regularly checks the MediaWiki about changes. This means that you must not always

[13] http://www.slideshare.net/

[14] http://www.slideshare.net/developers/documentation/

[15] Permalink is the unique URL of a MediaWiki page; e.g. `http://en.wikipedia.org/w/index.php?title=Java&oldid=366545644`

crawl and parse the complete MediaWiki, which would consume to much time. Only changes since the last successful crawling will be considered in a new crawling job.

General handling of jobs
All generated jobs will be stored in a threaded queue by scheduling first-come-first-serve. Each job is represented as a single thread, which allows to handle more than one job in parallel.

5 Outlook and further R&D opportunities

Artefact-Actor-Networks are analyzing interactions of learners with artefacts that are used for individual and organisational learning. The semantically enriched data is then exposes via an open API to be included in various user interfaces.

In [Reinhardt, 2010] we introduce the AANalyzer as the first awareness dashboard that build on the AAN model and will be applied to several learning communities in the course of the year 2010, which will help us to gain user feedback on the awareness support the tool offers. At the same time we are extending both the number of social media tools available for analysis in AANs as well as the quality of the AAN backend implementation. First functional tests with more than 400.000 nodes in an Artefact-Actor-Network revealed the need for improvements regarding the inferring of semantical data stored. Besides the long runtimes to calculate the inferred models, any changes in the data model require a rebuilding of the inferred model. Furthermore, the calculation of semantic similarity between artefacts, actors and groups in AANs is a challenging endeavor to overcome. At the moment the calculations are done in an online algorithm whose runtime is exponential to the number of artefacts in the DataStore. We strive for implementing an offline algorithm that makes use of caching strategies and a the data in the SimilarityStore.

Regarding the variety of awareness widgets for the users of the AANalyzer, we will extend the choice with statistics widgets and an advanced word cloud implementation that will allow for the visualisation of timely changes in the importance and use of certain terms.

References

[DCMI, 2010] DCMI. Dublin Core Metadata Initiative. `http://dublincore.org/`, 2010.

[Ebner and Schiefner, 2008] Martin Ebner and Mandy Schiefner. Microblogging - more than fun? In *Proceedings of the IADIS Mobile Learning Conference*, pages 155–159, 2008.

[Ellison *et al.*, 2007] N.B. Ellison, C. Steinfield, and C. Lampe. The benefits of Facebook" friends:" social capital and college students' use of online social network sites. *Journal of Computer Mediated Communication (Electronic Edition)*, 12(4):1143, 2007.

[FOAF, 2010] FOAF. The Friend of a Friend (FOAF) project. `http://www.foaf-project.org/`, 2010.

[Granovetter, 1983] M. Granovetter. The strength of weak ties: A network theory revisited. *Sociological theory*, 1:201–233, 1983.

[Gruber, 1993] T.R. Gruber. A Translation Approach to Portable Ontology Specifications. *Knowledge Acquisition*, 5(2):199–220, 1993.

[Heinze *et al.*, 2010] N. Heinze, P. Bauer, U. Hofmann, and J. Ehle. Kollaboration und Kooperation in verteilten Forschungsnetzwerken durch Web-basierte Medien – Web 2.0 Tools in der Wissenschaft. In *Forthcoming Proceedings of the GMW 2010 conference*, 2010.

[Java *et al.*, 2007] Akshay Java, Xiaodan Song, Tim Finin, and Belle Tseng. Why we twitter: Understanding microblogging usage and communities. In *Procedings of the Joint 9th WEBKDD and 1st SNA-KDD Workshop 2007*, August 2007.

[Lohmann and Riechert, 2010] S. Lohmann and T. Riechert. Adding Semantics to Social Software Engineering: (Re-)Using Ontologies in a Community-oriented Requirements Engineering Environment. In *Workshop-Proceedings of Software Engineering 2010*, pages 485–494, 2010.

[Maalej *et al.*, 2008] W. Maalej, D. Panagiotou, and H.-J. Happel. Towards Effective Management of Software Knowledge Exploiting the Semantic Web Paradigm. In *Proceedings of Software Engineering 2008*, pages 183–197, 2008.

[O'Reilly, 2005] T. O'Reilly. What is Web 2.0 – Design Patterns and Business Models for the Next Generation of Software. `http://oreilly.com/pub/a/web2/archive/what-is-web-20.html`, September 2005.

[Reinhardt *et al.*, 2009a] Wolfgang Reinhardt, Martin Ebner, Guenter Beham, and Cristina Costa. How people are using Twitter during conferences. In V. Hornung-Prähauser and M. Luckmann, editors, *Creativity and Innovation Competencies on the Web. Proceedings of the 5th EduMedia 2009, Salzburg*, pages 145–156, 2009.

[Reinhardt *et al.*, 2009b] Wolfgang Reinhardt, Matthias Moi, and Tobias Varlemann. Artefact-Actor-Networks as tie between social networks and artefact networks. In *Proceedings of the 5th International ICST Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom 2009)*, November 2009.

[Reinhardt, 2010] Wolfgang Reinhardt. A widget-based dashboard approach for awareness and reflection in online learning communities based on Artefact-Actor-Networks. In *Forthcoming Proceedings of the First PLE Conference 2010*, 2010.

[SIOC, 2010] SIOC. The semantically-interlinked online communities (sioc) project. `http://sioc-project.org/`, 2010.

[Vuorikari, 2009] Riina Vuorikari. *Tags and self-organisation: a metadata ecology for learning resources in a multilingual context*. PhD thesis, Open University of the Netherlands, 2009.