Kachako: A Hybrid-Cloud Unstructured Information Platform for Full Automation of Service Composition, Scalable Deployment and Evaluation

Natural Language Processing as an Example

Yoshinobu Kano

PRESTO, Japan Science and Technology Agency (JST), Japan
kano@kachako.org

Abstract. Automation is the key concept when designing a service platform, because automation could reduce human's work. Focusing on unstructured information such as text, image and audio, we implemented our service platform "Kachako" in a hybrid-cloud way where services themselves are transferred on demand. We suggest making each service specified by its input and output types, and executable of the service portable, compatible and interoperable. Assuming such services, Kachako thoroughly automates everything that users need. Kachako provides graphical user interfaces allowing end users to complete their tasks within Kachako without programming. Kachako is designed in a modular way by complying with well-known frameworks such as UIMA, Hadoop and Maven, allowing partial reuse or customization. We showed that Kachako is practically useful by integrating our natural language processing (NLP) services. Kachako is the world first full automation system for NLP freely available.

Keywords: Automation, Unstructured Information, Service Composition, Scalability, Natural Language Processing.

1 Introduction

One of the primary motivations providing services would be to save users' labor, i.e. *automation*. However, do current services sufficiently provide automation features? If we could thoroughly automate users' service related tasks, what remains as human's task? What do we need to achieve such automation?

Our answer is that the minimum user operations include only two steps: prepare a user account in machines, and specify a service that the user wishes to run. The rest of everything could be automated if we provide an ideal service platform. That said, unfortunately, existing service platforms tend to ask users too many manual tasks that could be potentially automated.

An advantage of web services would be the easiness of using services. However, web services, from users' point of view, have more or less fixed configurations both as software and hardware; services are under control of service providers, so

A. Ghose et al. (Eds.): ICSOC 2012, LNCS 7759, pp. 72–84, 2013.
© Springer-Verlag Berlin Heidelberg 2013

customization of already deployed services is limited; physical servers of services cannot be changed, which prevents service scalability and availability.

We could provide users the scalability and the availability by making the entire service deployment under control of users themselves. Such a control can be practically available if users could transfer the services themselves, but not the data to be processed, to arbitrary servers that users wish to use. We call this style as hybrid-cloud, because we take benefits of both cloud and local deployments. This is possible due to the recent growth of open source projects, including software from state-of-the-art research tools to enterprise middleware implementations. We exploit such freely available software, together with the decreasing cost of computational resources e.g. the so-called cloud servers, to make such portable services available.

In addition to the portability, services need to be compatible and interoperable to allow automatic service composition. We focus on unstructured information processing where we could assume relatively simpler input and output dependencies.

Assuming such services, a fully automated platform could be available. Kachako, our platform, is just such a full automation system. Kachako is publicly available under open source license[1]. Kachako is designed to thoroughly automate any procedure in using services for unstructured information processing: selection, composition, (parallel distributed) deployment, result visualization, and evaluation of services.

We would like to emphasize here that users of Kachako do not need to know any detail described in this paper, as these details are obscured due to the automation features. Even when users wish to customize our system, users simply have to learn a specific standardized interface which users are interested in.

In this paper, we first describe background and related works of this research in Section 2. Then we discuss ideal form of services from an automation point of view in Section 3. We describe our Kachako system architecture which automates total use of services in Section 4, details discussed in subsections. In Section 5, we describe our domain specific service implementation in natural language processing, showing that the architecture is practically available and useful integrating all of features described in this paper. Section 6 describes limitations of our architecture. We conclude this paper in Section 7, discussing possible future works.

2 Background and Related Works

We adopted Java Standard Edition 7 as the main programming language of our implementation, as Java is suitable to achieve portability over different environments.

Kachako is compliant with Apache UIMA [1]. UIMA, Unstructured Information Management Architecture, is a framework which provides metadata schemes and processing architecture [2]. We selected UIMA not just because we focus on unstructured information processing, but also UIMA is currently the most suitable open framework for the automation features we need; UIMA's block-wise architecture concept potentially offers easier service composition and scalability, although simply

[1] The Kachako system will be available from `http://kachako.org/`

using UIMA is not sufficient. UIMA's processing unit is called a UIMA component. We call a UIMA component as a service in this paper because we make any service into a UIMA component in Kachako. A UIMA workflow consists of a (nested) list of components. A parent component may have a programmable flow controller that decides the processing order of its child components. A collection reader is a special UIMA component which retrieves input data; a collection reader is normally located at the start of a workflow. UIMA-AS (asynchronous UIMA) is a set of UIMA's next generation architectures including UIMA-AS workflow, UIMA-AS web service, etc. UIMA's data structure is called CAS (Common Analysis Structure), which is represented as a Java object at runtime and normally stored as XMI (XML Metadata Interchange) format in the disk. A CAS consists of raw data part and annotations part. The raw data part is normally kept unchanged once stored, holds raw data of e.g. text, audio, image, etc. The annotations part holds directed graphs of feature structures, where some of the feature structures are linked with the raw data by data offset positions. This representation style is called a stand-off annotation style. Any feature structure should be typed by a user defined data type. Types are defined hierarchically in a type system XML file. UIMA-AS uses Apache ActiveMQ [3] (a reference implementation of JMS, Java Messaging Service) as a web server.

We also adopt other open source standards in Kachako. Regarding scalability middleware, we use Apache Hadoop [4] with HDFS. Hadoop is now very widely used and stable enough. Our service repository is based on Apache Ivy [5], which provides remote file fetching system with dependency descriptions either in its original format or in an Apache Maven [6] format.

There are many UIMA related works, while most of them only provide UIMA service components. The IBM's Watson Question-Answer (QA) system [7] is UIMA compliant, but Watson is specific to the QA (and answering the Jeopardy quiz) domain; Watson is commercial software and not publicly available. There are several UIMA compliant resources available [8][9] [10] but they are services not a platform.

There are also a couple of previous studies of workflow oriented systems, but previous discussions remained partial when seen from the automation point of view. U-Compare [11][12][13], our previous product, is a UIMA compliant platform but automation and scalability were insufficient. Taverna [14] is a workflow system widely used in the Bioinformatics domain. In Taverna users can connect web services in a graphical way but service compatibility and interoperability is not sufficiently considered, users need to understand each service behavior in detail and in most cases required to write a script to match their I/O formats. Galaxy [15] is another workflow based system. Galaxy's service I/O is simply files, so preferred by shell-based programmers. However, this easiness rather requires extra human works when connecting different services because each service may have different formats. Langrid [16] is a collection of NLP web services where a service administration system is provided. Langrid uses BPEL to describe workflows, assuming programming work to customize the workflows. GATE [17] is a total text mining programming environment like Eclipse, but is not intended to provide an automation platform like Kachako.

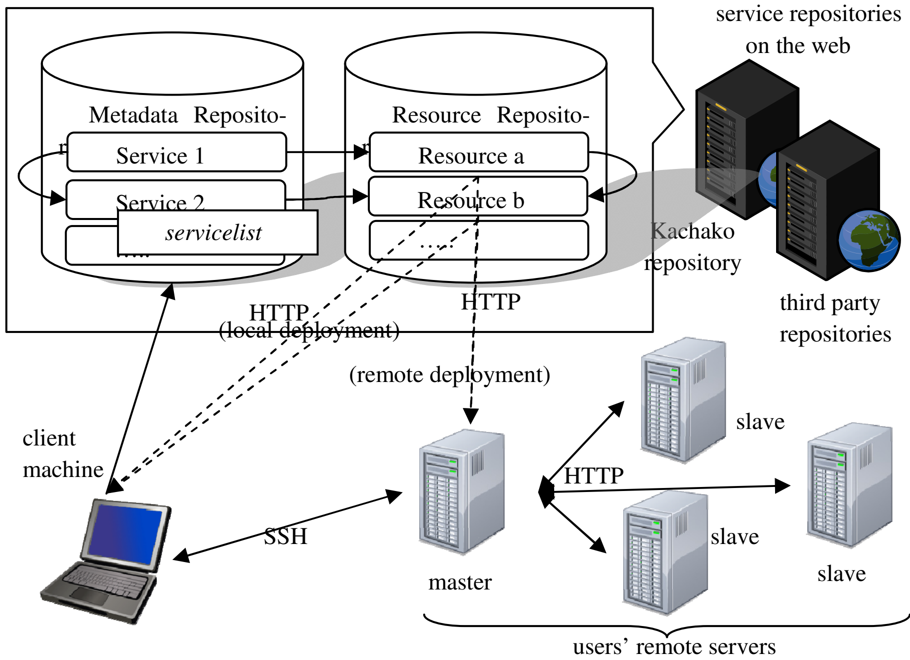

Fig. 1. A conceptual figure of Kachako's physical configuration. This figure illustrates a typical configuration, e.g. there may be no remote server used depending on users' configurations.

3 Forming Services for Automation

Users are often required expert knowledge in order to determine whether a pair of services can be composed or not. Such an interoperability issue depends not only on surficial format definition e.g. XML, but on deeper semantic compatibility. In a worse case, users need to re-implement the original service implementation for the services to be able to be combined. We certainly need automation here, as users are not necessarily programmers or experts. Furthermore, this is not an essential task for users.

Such service compatibility and interoperability problem includes several issues: data format, data type, service metadata, and form of services. As we adopted UIMA as the basic framework, so data format and service metadata description format are guaranteed to be compatible. We discuss the rest of theoretical issues in this section.

While standardization of metadata and data format syntax is often discussed, it tends to be missed in what shape a service should be formed. Some of the existing services are provided as APIs, while others are a large integrated application. From our point of view, reusability is the critical issue for the service users. If a service is smaller, there is more possibility to reuse the service; a smaller service could be more generic than a large application service, which would assume a more specific use case. However, this discussion of service granularity is not sufficient. While APIs (functions of programing languages) could be the smallest service we can provide,

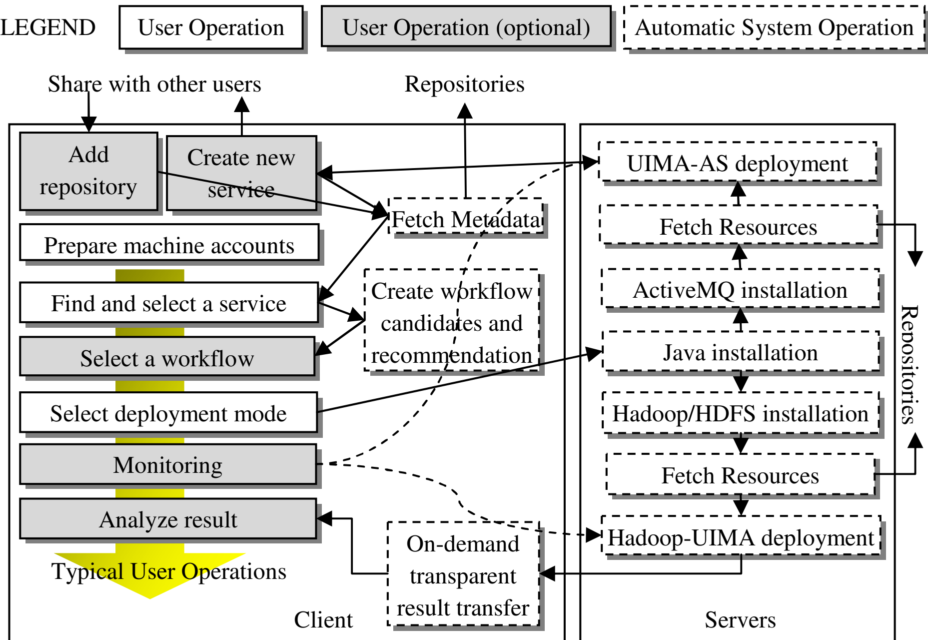

Fig. 2. A conceptual system flow diagram of Kachako's remote deployment

simply providing APIs do not achieve sufficient reusability. APIs are normally described by input data types and output data types. However, an API does not return meaningful result if we just combine arbitrary two API functions whose I/O data types formally match. This is because most of APIs have implicit conditions which are described in machine unreadable documentations, or often not described at all.

In order to allow automatic composition, we suggest that a service metadata should be specified by a list of I/O data types without any other implicit information. This obscures the implicit API conditions, so users and developers do not need to understand inside the service implementations. A service should be as small as possible at the same time, to be more reusable as discussed above. We can create such a service because tasks are composite in nature in case of unstructured information processing.

Scalability is another critical issue that we have to consider together with the reusability. If a service can process a block of input data without referring to another input block, then we can parallelize the service process by dividing the input into independent blocks. This parallelization can scale the entire process out without overheads of distributed communication cost. This is an ideal parallelization from the automation point of view because the original service can be reused without modification. Such block division can be decided by semantic relations in the original input data. For example, a document is normally independent in a collection of documents, while they are sometimes merged into a single file. The number of possible parallelization is decided by the number of blocks. Therefore, the input retrieval service, a collection reader in case of UIMA, should retrieve input by splitting the original data

into smallest but semantically independent blocks. If the input is split so, developers can implement a service without aware of such a scalability issue. In case a service needs to collect information over blocks, e.g. search engine indexing, the service should be specially implemented in a scalable way though this is limited to a couple of special purposes. We describe our ready-to-use services of NLP in Section 5.

4 Service Platform for Full Automation

Roughly speaking, Kachako physically consists of three parts: a *client* module including GUIs (Graphical User Interfaces), a *server* module, and repositories on the web. These modules can be used either in an integrated way or in a partial way; using the client only for lightweight tasks, using both client and server for automatic large-scale processing, using the server only to integrate with other existing systems, etc. We also provide modules in smaller granularity for the reusability.

Kachako's *client* runs in a machine where users can configure everything by GUIs. We only require Java 7 to be installed, so machines of any modern OS (Windows, Mac, Linux) are available. Most machines have Java pre-installed nowadays. Installation of Kachako's *client* is automatic. By running our small Java launcher program, all of required binaries and resources are downloaded, cached, and updated if there is any update. Kachako's *client* provides GUIs to configure workflows in an automated way, as described in the later sections. Kachako also provides a command-line mode, where users can run a specified workflow without the GUIs.

Fig. 1 illustrates architecture of the entire Kachako system conceptually. Fig. 2 shows a conceptual flow diagram of the system from the user's point of view.

4.1 Repository Architecture for Finding Services and Resolving Dependencies

Our goal in designing service repository architecture is that files can be shared efficiently, while dependencies between files could be automatically resolved.

For our dependency description, we adopt the Apache Ivy [5] format including Maven. Because Ivy allows specifying multiple repositories, it is possible to configure a cloned backup repository. Any resource is cached in the user's local disk. This mechanism allows efficient and dynamic resource distribution.

We separate our service repository into metadata and actual resources (executable binaries, external data etc.) for efficient data transfer. When users search and configure services into a workflow, they just need the metadata repository. After creating a workflow, the Kachako system can collect required resources assuming resource dependencies are defined properly. Fig. 1 illustrates this architecture conceptually.

Third party service providers can distribute their services by building their own repository. In addition to our default service repository, users can add such a third party repository by simply specifying a repository location URL as described below. Then Kachako seeks for available services and resolves any required dependencies automatically.

4.2 Workflow Oriented Automatic Service Composition

When services are formed as described in the previous section, all of possible combinations of services can be theoretically calculated from services' I/O conditions. Kachako considers data type hierarchy which makes this calculation a bit complex. The entire combinations of services form a directed graph structure in general.

The number of the combinations may become too large for humans to grasp, so a proper filtering feature would be helpful. Because users' goal is usually linked with the final output, Kachako asks users to specify which service they wish to run as the final output service. Kachako also asks users to specify which collection reader to retrieve as workflow input. These are the only decisions of users, which a system cannot automatically determine. Given these input and output services, Kachako's automatic workflow composition GUI shows possible service combinations. Users can further filter workflows by specifying intermediate services, while hasty users can immediately run a suggested workflow.

Kachako provides another workflow creation GUI. Users can specify components one by one manually in a dragging-and-dropping manner, where any UIMA workflow can be created even ignoring the I/O conditions.

Kachako further provides other automatic service composition GUI for comparison and evaluation as described in Section 4.4.

4.3 Automatic Service Deployment, Execution and Monitoring

Kachako provides three service deployment modes: *local*, *batch*, and *listener* deployment mode. Most of users' requirements can be satisfied by using one of, or a combination of these modes. Details are described in subsections.

4.3.1 Local Deployment: Automatic Service Deployment in Local Machine

The local deployment would be suitable for running lightweight tasks immediately. We assume that service metadata and resources are properly configured as described in Section 3. By tracing the dependency information for each service in a workflow,

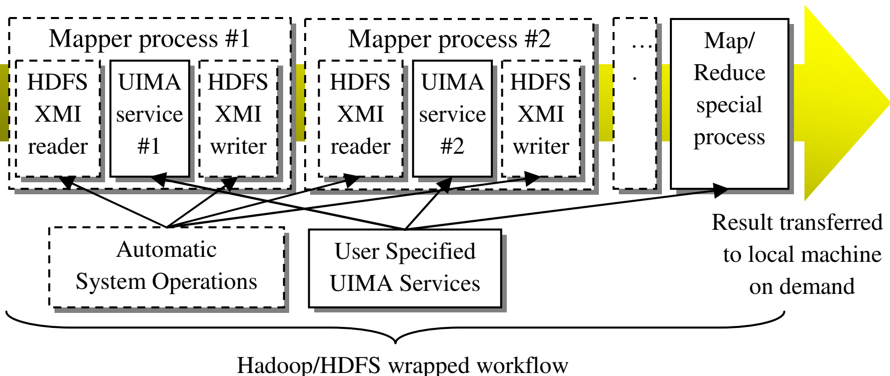

Fig. 3. A diagram of Hadoop-UIMA processing

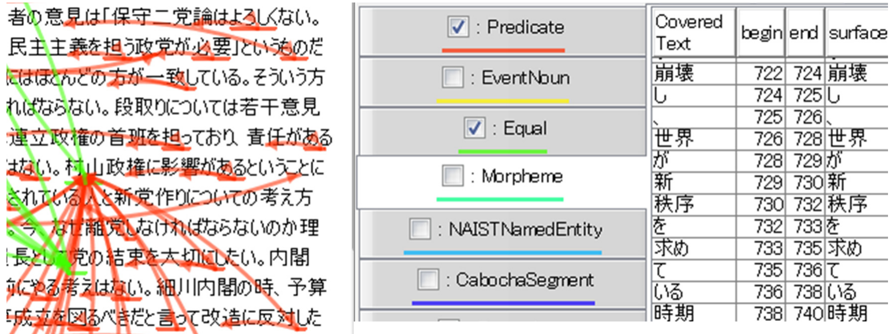

Fig. 4. An example of NLP information visualizing co-reference and predicate relations (left) and showing details as a table (right)

Kachako can obtain all of required resources. Each service is assigned a separate Java class loader to avoid version conflicts of libraries, which often occur and difficult to resolve. This is common with other deployment modes. Kachako deploys a specified workflow as a UIMA-AS service locally and saves its result in UIMA's XMI format.

4.3.2 Automatic Server Configuration for Batch and Listener Modes

For the other two modes, remote server configuration is required. We aim to avoid any permission of root authority that becomes a bottleneck in the setup tasks. Kachako's only requirement is that a user should prepare Linux based machines where the user has his/her user account, accessible via SSH and connected to the Internet via HTTP. Once the account is registered, Kachako sets up everything automatically. If a machine configuration is provided by administrator using our importable configuration format, users can skip this registration step.

Kachako uses user's home directory or the OS's temporary directory as its root directory for saving any file. In case of the temporary directory, Kachako prevents the OS to delete the saved files. Kachako automatically installs Java 7 if not installed yet.

4.3.3 Batch Deployment: Remote Batch Scalable Processing with Hadoop

The batch deployment uses Apache Hadoop [4] as a low level API. We assume that a workflow, which a user wishes to run, is given as described in the previous sections.

Installation and deployment of Hadoop are not so easy task for end users. Kachako automatically installs and deploys Hadoop/HDFS. Recent multi-core servers are not efficiently used because required parallelization works are not essential for the users; users just want to scale out. Kachako's automatic scalable deployment would increase the efficiency. Users can customize configurations when they need specific tuning.

Kachako's server configuration GUI allows users to create a Hadoop/HDFS cluster setting from the registered servers. This setting includes Hadoop's JobTracker, TaskTracker(s), NameNode, and DataNode(s). Using our modules described earlier, Kachako automatically installs Java and Hadoop by creating relevant configuration files for each server in accordance with the user's server setting.

Performance Statistics | Annotation Statistics | Token Comparison

Assumed Gold Standard		Comparison Components		Total (All Documents)					
(.Sentence)	(.Sentence) .Token	(.Sentence)	(.Sentence) .Token	G	T	M	F1	PR	RC
UIMA	UIMA	UIMA	OpenNLP	97	81	71	79.78	87.65	73.20
UIMA	UIMA	UIMA	Genia with Tokenization	97	89	86	92.47	96.63	88.66
UIMA	OpenNLP	UIMA	UIMA	81	97	71	79.78	73.20	87.65
UIMA	OpenNLP	UIMA	Genia with Tokenization	81	89	74	87.06	83.15	91.36
UIMA	Genia with Tokenization	UIMA	UIMA	89	97	86	92.47	88.66	96.63
UIMA	Genia with Tokenization	UIMA	OpenNLP	89	81	74	87.06	91.36	83.15

Fig. 5. An example of evaluation statistics comparing three different tokenizers. Left hand columns show service names, right hand numerical value columns show standard evaluation statistics in NLP (F1, precision and recall scores).

Then Kachako will deploy the given UIMA workflow. If there is any required resource created locally, Kachako archives these resources, creates a local Ivy repository, and transfers them over SSH. A collection reader will run first to retrieve input data into HDFS in the XMI format. For each service in the rest of the top level services in the workflow, Kachako runs a Hadoop's Mapper without Reducer. As illustrated in Fig. 4, our special XMI reader and writer transfers XMI files from and to the HDFS file system. These allow mostly any UIMA service to be deployed in Hadoop/HDFS without modifying the original service implementation. Some special services, such as search engine indexing, would need Map/Reduce implementation.

Finally, Kachako provides on-demand result transfer feature. After finishing the workflow in Hadoop, Kachako transfers an index and statistics of result files to the client. When users need the content of the result e.g. for visualization, Kachako transfers relevant files from remote HDFS to the client in an on-demand way.

A monitoring feature is important for users, especially because large scale processing could take very long time, sometimes fails due to unpredictable reasons. Kachako shows Hadoop job's progress status in the GUI, as users' primary concern would be how much the job has progressed, and whether they are dead or alive. Further monitoring information can be shown simply by clicking a button in our GUI.

4.3.4 Listener Deployment: Scalable Remote Deployment with UIMA-AS

The listener deployment mode satisfies broad range of use cases. Firstly, although we assume freely available software, there would be certain requests not to provide source codes but services only. Secondly, some services may take very long time to initialize, or require special environment to run e.g. very large amount of disk space, difficult to setup for end users, etc. We can avoid such problems by deploying specific services as web services using our listener deployment mode.

By the UIMA-AS web service, we can deploy any local UIMA component as a web service. Kachako installs ActiveMQ in a specified remote server first. Then Kachako deploys specified services as UIMA-AS services. Required resources are transferred as same as the Hadoop mode above.

Simply deploying as a UIMA-AS web service does not scale. We provide a load balancer which distributes requests over UIMA-AS service nodes, pretending as a scalable single UIMA-AS service as a whole. Users can deploy such a scalable

UIMA-AS cluster by specifying a load-balancer server and slave servers for the UIMA-AS services. Users can deploy, undeploy, and monitor services via the GUI.

4.4 Automatic Service Evaluation by Combinatorial Workflow Composition

Because there are many similar but different services available, comparison and evaluation of services are critical issues. Services are more or less black-boxed, and behave differently depending on their input. Thus it is impossible to predict the best combination of services for a specific goal without actually running services.

As we discussed in Section 4.2, possible combinations of services can be calculated. Because such combinations tend to share partial graph, we can efficiently run combinations of services rather than separately running each of combinations as independent workflows. The basic concept is similar to our previous work [11], but in the previous work we assumed manual configurations which were difficult for users to configure. In contrast, Kachako automates everything by a new architecture as below.

Firstly, Kachako calculates a possible service combination graph as described in Section 4.2. Then, for each edge of the graph, CAS content is filtered by the input type(s), copied to a new CAS and passed to the next service. After processing the next service, output of the service is internally grouped and stored back into the CAS. This architecture allows an efficient automatic execution of combinatorial workflows, while the original services do not need to be modified.

By plugging comparison metric services, users can obtain statistical values for each pair of comparable service graphs. If a pair includes the so-called gold standard data, i.e. the correct answer, then the comparison becomes an evaluation. Fig. 5 shows an example comparison result for an NLP task, tokenization.

The above discussion raises an issue, in what way data types should be defined. Our automatic service composition is based on the I/O metadata descriptions of services, which are described in terms of data types. Therefore, data types should include, at least, types which are used to describe the I/O metadata. In addition, data types should include concepts which are used to compare and evaluate services as discussed in this section. Actual data type definition is a domain specific issue.

5 Ready-to-Use Implementation for Natural Language Processing as System Evaluation

The Kachako platform architecture we discussed so far is generic. However, we claim that an ideal system should help users by automation as much as possible. It is absolutely required to provide actual implementation for a specific target domain; else the system would be just useless as it is too abstract. We show and evaluate our system's usefulness by implementing domain specific parts of the system, for the NLP domain of text processing.

Domain specific issues include data visualization, data type definition, and actual services. Our system assumes a trial-and-error style use case, in order for users to obtain the most suitable workflow. Thus error analysis, especially the visualization

Table 1. Result of performance test in the batch mode. Input is the BioMedCentral's full text corpus. # of input is the documents processed, # of mappers is parallel process counts, actual time is elapsed time for the processes, total CPU time is sum of CPU time over mappers.

# of Input	20	20	100	100	1000	1000
# of Mappers	5	10	5	10	5	10
Actual Time (s)	130	61	584	379	5684	3666
Total CPU Time (ms)	508,720	68,870	2,442,520	2,483,580	23,963,340	23,959,530

feature, is very important. We have developed a generic visualizer for text which can show annotations and relations of annotations graphically (Fig. 6).

Developing services and defining data types are not a separate issue. We have been developing compatible NLP services from basic linguistic tools to applied text mining tools in different languages. We also provide utilities to help developers wrap existing tools into compatible UIMA services. Everything is integrated into the Kachako system, allowing users to find an NLP service, create and run a workflow, and analyze its result in an ultimately automated way. The number of our services is currently around one hundred, which can generate thousands of possible workflows theoretically.

We have performed a scalability test by using the NLP services. As a realistic scenario, we used the BioMedCentral's full text corpus [18] as input and performed a protein mention extraction task by ABNER [19] in our batch processing mode. Table 1 is the statistics of the testing. Some overhead was observed as expected, but it scaled out as a whole when increasing the number of mappers.

6 Limitation

One of the limitations is authentication. There would be certain needs for user authentication. Our listener deployment could provide authentication of services. The component repository could also limit users. However, these are not supported currently.

Another limitation is the way forming services. Unfortunately, not all the services can be ideally formed like we discussed. For example, dictionaries are often used as external resources in NLP tools. Although it is ideal for such external resources to be compatible, we currently simply specify locations of resources. Such resources are read in the initialization time but not read during the process time, and so it is unnatural and difficult to put the resource into the CAS.

The other type of limitations is stability and compatibility of the data type definitions. We have been implicitly assuming that data types are static. However, if an incompatible type system is used, previously created services and their results become incompatible. A solution would be to develop a type system converter. But it is not a trivial task as there could be many incompatible type systems by different developers.

7 Conclusion and Future Work

In this paper, we proposed architecture to ultimately automate tasks using services, and showed its implementation is practically useful, in the NLP domain as an example. This system, Kachako, is the world first system providing such thorough automation features in a scalable and reusable way; select a service and specify servers to run, that's all. Board range of standards and technologies were harmonized for these automation features to be reusable. Increasing the number of available services, including Map/Reduce services, is the future work. Enhancement of the Kachako system to support other domains would be a future work as well.

Acknowledgements. This work was partially supported by JST PRESTO and KAKENHI 21500130 (MEXT, Japan).

References

1. Apache UIMA, `http://uima.apache.org/`
2. Ferrucci, D., Lally, A., Gruhl, D., Epstein, E., Schor, M., Murdock, J.W., Frenkiel, A., Brown, E.W., Hampp, T., Doganata, Y., Welty, C., Amini, L., Kofman, G., Kozakov, L., Mass, Y.: Towards an Interoperability Standard for Text and Multi-Modal Analytics. IBM Research Report, RC24122 (2006)
3. Apache ActiveMQ, `http://activemq.apache.org/`
4. Apache Hadoop, `http://hadoop.apache.org/`
5. Apache Ivy, `http://ant.apache.org/ivy/`
6. Apache Maven, `http://maven.apache.org/`
7. Ferrucci, D.A.: Introduction to This is Watson. IBM Journal of Research and Development 56, 1:1–1:15 (2012)
8. Hahn, U., Buyko, E., Landefeld, R., Mühlhausen, M., Poprat, M., Tomanek, K., Wermter, J.: An Overview of JCoRe, the JULIE Lab UIMA Component Repository. In: LREC 2008 Workshop, Towards Enhanced Interoperability for Large HLT Systems: UIMA for NLP, Marrakech, Morocco, pp. 1–8 (2008)
9. Hernandez, N., Poulard, F., Vernier, M., Rocheteau, J.: Building a French-speaking community around UIMA, gathering research, education and industrial partners, mainly in Natural Language Processing and Speech Recognizing domains. In: LREC 2010 Workshop of New Challenges for NLP Frameworks, Valletta, Malta (2010)
10. Ogren, P.V., Wetzler, P.G., Bethard, S.: ClearTK: A UIMA Toolkit for Statistical Natural Language Processing. In: LREC 2008 Workshop 'Towards Enhanced Interoperability for Large HLT Systems: UIMA for NLP', Marrakech, Morocco, pp. 32–38 (2008)
11. Kano, Y., Miwa, M., Cohen, K., Hunter, L., Ananiadou, S., Tsujii, J.: U-Compare: a modular NLP workflow construction and evaluation system. IBM Journal of Research and Development 55, 11:1–11:10 (2011)
12. Kano, Y., Dorado, R., McCrohon, L., Ananiadou, S., Tsujii, J.: U-Compare: An Integrated Language Resource Evaluation Platform Including a Comprehensive UIMA Resource Library. In: 7th International Conference on Language Resources and Evaluation (LREC 2010), Valletta, Malta, pp. 428–434 (2010)

13. Kano, Y., Baumgartner, W.A., McCrohon, L., Ananiadou, S., Cohen, K.B., Hunter, L., Tsujii, J.: U-Compare: share and compare text mining tools with UIMA. Bioinformatics 25, 1997–1998 (2009)
14. Hull, D., Wolstencroft, K., Stevens, R., Goble, C., Pocock, M.R., Li, P., Oinn, T.: Taverna: a tool for building and running workflows of services. Nucleic Acids Res. 34, W729–W732 (2006)
15. Blankenberg, D., Von Kuster, G., Coraor, N., Ananda, G., Lazarus, R., Mangan, M., Nekrutenko, A., Taylor, J.: Galaxy: a web-based genome analysis tool for experimentalists. Curr. Protoc. Mol. Biol. ch. 19, Unit 19.10.1–19.10.21 (2010)
16. Ishida, T.: Language Grid: An Infrastructure for Intercultural Collaboration. In: Proceedings of the International Symposium on Applications on Internet, pp. 96–100. IEEE Computer Society (2006)
17. Cunningham, H., Maynard, D., Bontcheva, K., Tablan, V.: GATE: A framework and graphical development environment for robust NLP tools and applications. In: 40th Anniversary Meeting of the Association for Computational Linguistics, Philadelphia, USA, pp. 168–175 (2002)
18. BioMed Central's open access full-text corpus, `http://www.biomedcentral.com/about/datamining`
19. Settles, B.: ABNER: an open source tool for automatically tagging genes, proteins and other entity names in text. Bioinformatics 21, 3191–3192 (2005)