SIA: Scalable Interoperable Annotation Server

Johannes Kirschnick and Philippe Thomas

DFKI Language Technology Lab
DFKI Intelligent Analytics for Massive Data Lab
Alt-Moabit 91c, 10559 Berlin, Germany

Abstract. Recent years showed a strong increase in biomedical sciences and an inherent increase in publication volume. Extraction of specific information from these sources requires highly sophisticated text mining- and information extraction-tools. However, the integration of freely available tools into customized workflows is often cumbersome and difficult. We describe *SIA*, our contribution to the BeCalm-TIPS task, a scalable, extensible, and robust annotation service. The system currently covers three named entity types (*i.e.,* Mutations, Diseases, and miRNA) and is freely available under Apache 2.0 license at `https://github.com/Erechtheus/sia`.

Key words: Annotation service, Robustness, Scalability, Extensibility

1 Introduction

A vast amount of information on biomedical processes is scattered over millions of scientific publications. Manual curation of this information is expensive and cannot keep up with the ever increasing volume of biomedical literature [7]. To this end, several sophisticated natural language processing tools have been proposed to assist professionals in finding specific information from texts. Many of these highly specialized tools are provided as open source projects to the community. However, the integration of state-of-the-art open source tools into customized text-mining workflows is often difficult and cumbersome [10,12]. Standardized interchange formats, such as BioC [5], enable the exchange of text mining results but the initial set-up of these tools is still an unsolved issue. Exposing tools via public web services implementing common specifications bypasses this problem and allows a code-agnostic integration of specific tools by providing an interoperable interface to third parties. This enables simple integration, comparison, and aggregation of different state-of-the-art tools. In this publication we present *SIA*, our contribution to the BeCalm TIPS task [9], a robust, scalable, extensible, and generic framework to combine multiple named entity recognition tools into a single system.

The publication is organized as follows: First, we provide a general description of the system architecture, followed by details concerning the implementation and failure handling, and end with a summary and future work section.

2 General Architecture

Design Goals: *SIA* is designed around the following three main concepts:

1. **Scalability** Ability to handle a large amount of concurrent requests, tolerating bursts of high request rates over short periods of time.
2. **Robustness** Temporary failures (*e.g.,* networking problems or server failure) should be handled transparently and not lead to dropped requests.
3. **Extensibility** Enable simple integration of arbitrary NLP tools to reduce initial burden for providing an annotation service.

Figure 1 shows a high level overview of the general architecture. Overall, *SIA* consists of three logical parts, the front end, back end as well as a result handling component. A message based architecture and route handling based on Enterprise Integration Patterns specifies how requests are handled by different components and flow through the system. While a complete discussion of the integration patterns is out of scope of this publication, interested readers can refer to [6] for a detailed description.

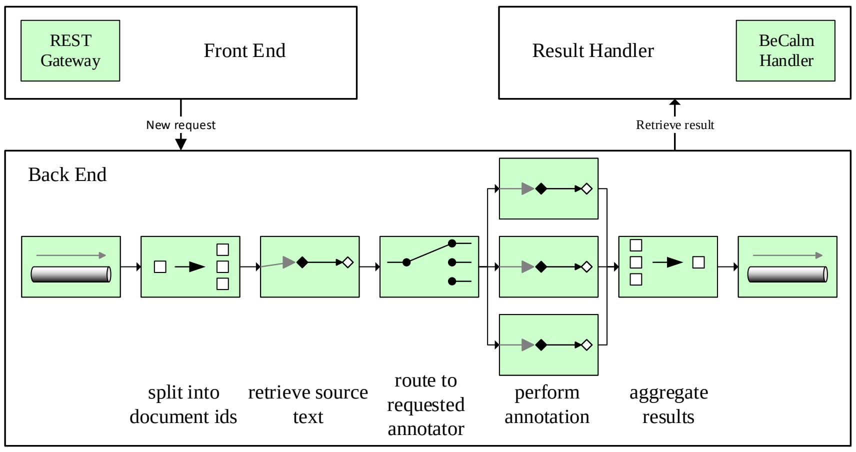


Fig. 1: General Architecture of *SIA*. The front end handles new requests and forwards them to the back end over a message bus. Each message is transformed through a series of components, which in turn are connected via named queues. The result handler collects the annotation responses and returns them to the calling client.

The **front end** is the user facing component, handling incoming annotation requests. Received requests are forwarded to the **back end**, which downloads individual documents and feeds them through a chain of annotators. The results are made available to the **result handler**, which sends the annotations back to the requester.

All components are connected to a message bus over which they exchange messages. This loosely coupled design allows to easily scale, replace and augment each participant in the message flow independently. Persistent named queues are defined as input and output for all components. These queues are stored for the entirety of the systems lifetime. This architecture provides fault tolerant and scalable processing. Fault tolerance is enabled through component wise acknowledgment of each successful message processing, which allows replaying all unacknowledged messages during system recovery.

Messages carry information through the system and consist of a HEADER and PAYLOAD part. The HEADER contains meta information, such as expiry date, global ids or requested annotation types and is used by the system to route messages to the respective consumers. The PAYLOAD contains the actual data to be processed. Each requests is translated into a new message that flows through the system, is enhanced, transformed and aggregated by parts of the message flow to derive a annotation result.

The following sections describe each individual system component in details.

2.1 Front end

The front end encapsulates the annotation processing from the clients and serves as the entry point to the system. Currently it provides a REST endpoint according to the Becalm-TIPS task specification[1]. Incoming requests are translated into messages and forwarded to the input queue. This way, the overall processing in the front end is very lightweight and new requests can be handled irrespectively of any ongoing annotation processing.

To handle multiple concurrent requests with varying deadlines, we make use of the fact that the input queue is a priority queue, and prioritize messages with an earlier expiry date[2]. The message expiry date, as provided by the calling clients, is translated into a message priority. Using the currently processed messages and their deadlines as well as past elapsed processing time statistics are used to estimate the individual message urgency.

The front end also handles validation and authorization, which moves this logic into a central place. To monitor the system, statistics are served about average requests rate, document types as well as back end processing counters.

2.2 Back end

The back end is concerned with fetching documents from the supported resources, calling the requested annotators for each resulting text fragment, aggregating the results and feeding them to a result handler.

The back end process is modeled using a sequence of message transformations, which subsequently read from message queues and post back to new ones. The message flow starts by reading new requests from an input queue. As a

[1] Other entry points can easily be added

[2] Already running requests will not be canceled, the priority is just used as a fast path

single annotation request consists of multiple document ids, incoming messages are first split. Splitting takes one message as input and generates as many individual messages as there are document ids specified. Each document id is then retrieved by passing it through a corpus adapter, which fetches the raw text from the respective endpoint. The outcome is the retrieved text separated into abstract and title.

Texts are delivered to registered annotators using a recipient list. As annotators have a dedicated input queue in the system, each message header is inspected for any requested annotation types, forwarding the message to all matching queues. This design allows to easily add new annotation components by registering a new input queue and adding it to the routing endpoint candidates. All annotators forward their results to the same result queue, where they are collected and aggregated. Aggregation is the reverse of splitting and combines all annotation results into a single message. The aggregation key is a unique request id, set by the front end and stored in the message header. Finally, the aggregated message is forwarded to an output queue.

While the processing flow is specified in a sequential manner, this does not entail single threaded execution. Each individual transformer, such as a download adapter or an annotator, works independently and can be further scaled out. For example, having more than one annotator of the same type (potentially spread across multiple machines) can be used to increase document throughput. Furthermore, multiple requests can be handled in parallel at different stages of the pipeline. Fault tolerance is achieved by transacting the message delivery to each transformer and retrying on failure. Overall, the back end specifies an ordered execution flow and provides two injection points where users can add new functionality with additional corpus adapters or new annotation handlers.

It is noteworthy to point out, that annotation handlers can be hosted inside of *SIA*, or externally, which enables to integrate annotation tools cross programming languages and operating systems.

2.3 Result handler

Aggregated annotation results from the back end are committed to the message bus and picked up for further processing. In the current design only one result handler is specified. We implemented a REST handler according to the TIPS task definition, which posts annotation results back to the requester. Additional consumers, such as statistics gatherer or result archival processes can easily be added.

3 Implementation

SIA is implemented in Java and uses RabbitMQ[3] as message bus implementation. To exemplify the extensibility of our approach, we integrated NER components for three different entity types: Mutation names are extracted using

[3] `https://www.rabbitmq.com/`

Listing 1.1: Extension interface definition for annotators

```
public interface Annotator {
    Set<PredictionResult> annotate(ParsedInputText payload);
}
```

Listing 1.2: Extension interface definition for corpus adapter

```
public interface DocumentFetcher {
    List<ParsedInputText> load(IdList idList);
}
```

SETH [11]. For micro-RNA mentions we implement a set of regular expressions[4], which follow the recommendations for micro-RNA nomenclature [3]. Disease names are recognized using a dictionary lookup [1], generated from UMLS disease terms [4].

The messaging abstraction provides a clean separation of message routing and the actual message processing. Listing 1.1 shows the general interface contract *SIA* is expecting for each annotator, which does not expose the messaging infrastructure. Thus integrating the aforementioned annotators is as simple as implementing three wrappers and registering them as routing endpoints.

To increase the throughput of the back end, multiple instances of *SIA* can be started on different machines, where each instance would process requests in a round robin fashion. If this scaling is coupled with an input queue monitoring, the back end can be automatically scaled up or down to respond to changes in load pattern. Instead of scaling the complete back end, individual components can also be duplicated in a similar fashion, if they present a processing bottleneck.

3.1 Corpus Adapters

SIA contains corpus adapters for PubMed, PMC, Patent server and the BeCalm abstract server. These components are represented as transformers, which process document ids and return retrieved source texts. They are implemented following the interface definition shown in Listing 1.2, which similarly does not expose the messaging infrastructure. If an adapter supports bulk fetching of multiple documents, we feed a configurable number of ids in one invocation.

Since connecting to these endpoints is effectively calling a potentially unreliable remote service over an unreliable channel, retry on failure is used. This is backed up by the observation, that the most commonly observed error was a temporarily unavailable endpoint. To spread retries we use exponential backoff on continuous failures with an exponentially increasing time interval, capped at a maximum (initial wait $1s$, multiplier 2, max wait $60s$). If an endpoint fails

[4] `https://github.com/Erechtheus/mirNer`

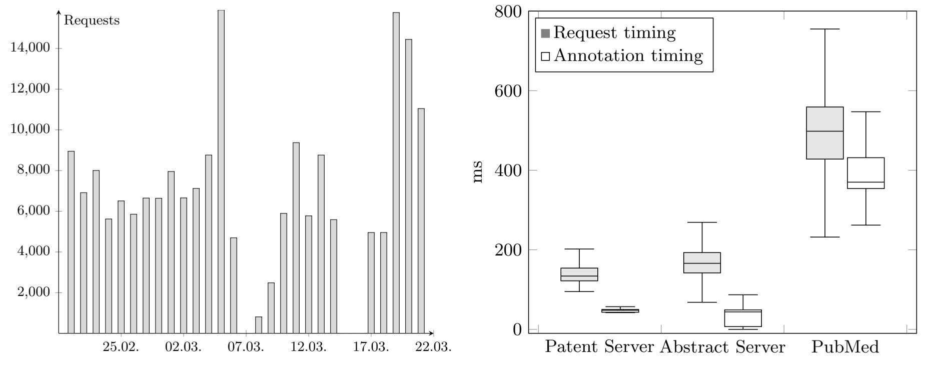
to respond after a configurable number of retries, we mark that document as unavailable and continue the processing. This allows a trade-off between never processing any results and giving up too early.

4 Failure Handling

In the following we describe the strategies implemented in *SIA* for dealing with errors.
Invalid requests Invalid requests represent client calls with wrong or missing information. These are handled in the front end using request validation and are communicated back to the caller with detailed error descriptions.
Backpressure To avoid that a large number of simultaneous requests can temporarily overload the processing system, *SIA* buffers all accepted requests in the input queue - using priorities to represent deadlines. Processing components can be scaled up or down by attaching more back end instances.
Front end fails If the front end stops, new requests are simply not accepted, irrespective of any ongoing processing in the back end.
Back end unavailable Messages are still accepted and buffered when there is enough storage space, otherwise the front end denies any new annotation requests.
Back end fails If the back end stops while there are still messages being processed, these are not lost but retried upon restart. This is enabled by acknowledging each message only upon successful processing per component.
Corpus adapter fails Each adapter retries, using exponential backoff, to fetch a document before it is marked as unavailable. As the BeCalm-TIPS task does not specify how to signal unavailable documents, these are just internally logged. Any subsequent processing treats a missing document as one with no content.
Annotator fails If an annotator fails on a particular message, this can potentially harm the entire back end when annotators are embedded in the system. As annotators are software components not under the control of the processing pipeline, we catch all recoverable errors and return zero found annotations in these cases - logging the errors for later analysis.
Result Handling fails The TIPS task description expects the result of an annotation request to be delivered to a known endpoint. If this fails, it is retired in a similar manner to the corpus adapter failure handling.
Message expired Clients can define a time until when a processing has to be finished. This is mapped to a time-to-live attribute of each message. This results in automatically dropping any expired message from the message bus.

5 Runtime

SIA is very lightweight and runs anywhere there is a Java environment and a connection to RabbitMQ available. Annotators can be directly embedded or configured to run externally, exchanging messages through the bus. We deployed



(a) Daily request rates

(b) Requests processing times for different endpoints

Fig. 2: Processing statistics over a four week period and request times per corpus, reporting complete processing and annotation timings separately.

SIA into Cloud Foundry, a platform as a service provider, which enables deployments of cloud components [8]. The front end and back end are deployed as two separate application containers. To ease development and running the service, we used a continuous integration workflow. Any code changes automatically trigger a redeployment of the service upon successful test runs.

Figure 2 shows that our system is capable of sustaining a high number of daily requests. Furthermore we observed that the request handling is dominated by corpus downloading times, which make up about 50% of the overall request time. This validates our decision to support bulk downloading of requests, as this amortizes the networking overhead over a number of documents. PubMed articles tend to be longer and thus incur higher annotation times. We also estimated the message bus overhead to about 10%, stemming from individual message serialization and persistence.

6 Summary and Future Work

We described *SIA* our contribution to the BeCalm-TIPS task which provides **scalability** - through component replication, **fault tolerance** - through message acknowledgement, and **extensibility** - through well defined injection points – with a particular emphasis on failure handling. The message bus provides a good design blueprint, which can be augmented with additional components. One interesting further development path is to port *SIA* to a distributed streaming environment such as Flink [2] or Spark [13]. These systems reduce the overhead of the message bus at the expense of more complex stream handling and aggregation. While many of the existing components could be reused, most engineering would need to be spent on implementing a fault tolerant window aggregation.

To encourage further discussion, the source of our current solution is freely available under Apache 2.0 license at `https://github.com/Erechtheus/sia` along with detailed descriptions on how to run and deploy the system.

Acknowledgments. This research was partially supported by the German Federal Ministry of Economics and Energy (BMWi) through the projects MACSS (01MD16011F), SD4M (01MD15007B) and by the German Federal Ministry of Education and Research (BMBF) through the project BBDC (01IS14013E).

References

1. Aho, A.V., Corasick, M.J.: Efficient String Matching: An Aid to Bibliographic Search. Commun. ACM 18(6), 333–340 (1975)
2. Alexandrov, A., Bergmann, R., Ewen, S., Freytag, J.C., Hueske, F., Heise, A., Kao, O., Leich, M., Leser, U., Markl, V., et al.: The Stratosphere platform for big data analytics. The VLDB Journal 23(6), 939–964 (2014)
3. Ambros, V., Bartel, B., Bartel, D.P., Burge, C.B., Carrington, J.C., Chen, X., Dreyfuss, G., Eddy, S.R., Griffiths-Jones, S., Marshall, M., Matzke, M., Ruvkun, G., Tuschl, T.: A uniform system for microRNA annotation. RNA 9(3) (2003)
4. Bodenreider, O.: The Unified Medical Language System (UMLS): integrating biomedical terminology. Nucleic Acids Res. 32(Database issue), D267–270 (2004)
5. Comeau, D.C., Doğan, R.I., Ciccarese, P., Cohen, K.B., Krallinger, M., Leitner, F., Lu, Z., Peng, Y., Rinaldi, F., Torii, M., et al.: Bioc: a minimalist approach to interoperability for biomedical text processing. Database 2013 (2013)
6. Hohpe, G., Woolf, B.: Enterprise integration patterns. In: 9th Conference on Pattern Language of Programs. pp. 1–9 (2002)
7. Hunter, L., Cohen, K.B.: Biomedical language processing: what's beyond pubmed? Mol Cell 21(5), 589–594 (2006)
8. Kirschnick, J., Alcaraz Calero, J.M., Goldsack, P., Farrell, A., Guijarro, J., Loughran, S., Edwards, N., Wilcock, L.: Towards an architecture for deploying elastic services in the cloud. Softw. Pract. Exper. 42(4), 395–408 (2012)
9. Pérez-Pérez, M., Pérez-Rodríguez, G., Blanco-Míguez, A., Fdez-Riverola, F., Valencia, A., Krallinger, M., Lourenco, A.: Benchmarking biomedical text mining web servers at BioCreative V.5: the technical Interoperability and Performance of annotation Servers - TIPS track. In: Proceedings of the BioCreative V.5 Challenge Evaluation Workshop. pp. 12–21 (2017)
10. Rheinländer, A., Lehmann, M., Kunkel, A., Meier, J., Leser, U.: Potential and pitfalls of domain-specific information extraction at web scale. In: Proceedings of the 2016 International Conference on Management of Data. pp. 759–771 (2016)
11. Thomas, P., Rocktäschel, T., Hakenberg, J., Lichtblau, Y., Leser, U.: SETH detects and normalizes genetic variants in text. Bioinformatics 32(18), 2883–2885 (2016)
12. Thomas, P., Starlinger, J., Leser, U.: Experiences from Developing the Domain-Specific Entity Search Engine GeneView. In: Proceedings of Datenbanksysteme für Business, Technologie und Web. pp. 225–239 (2013)
13. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: Cluster computing with working sets. In: Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing. pp. 10–10. Berkeley, USA (2010)