Architectural Backbone Evaluation for Data Stream Processing within the WINNER DataLab A Project Focused Point of View

Sebastian Apel*, Florian Hertrampf†, and Steffen Späthe‡

Chair for Software Engineering
Friedrich-Schiller-University Jena
D-07745 Jena, Germany
Email: sebastian.apel@uni-jena.de*, florian.hertrampf@uni-jena.de†, steffen.spaethe@uni-jena.de‡

Abstract—**Smart platforms for the integration of sensor and actor networks require collecting, analysing and evaluating data. Our research project WINNER aims to integrate systems which cover electromobility, energy consumption within residential areas, local energy production, e. g., with photovoltaic systems, and storages for local smart grids. WINNER wants to use such a platform to optimise the energy consumption. Every actor within a residential area has to be considered, and integration into a centralised data stream process is necessary. As a non-hard real-time system, the platform has to solve enterprise application integration problems, looking at complex event processing and knowledge discovery in data. This paper targets to analyse possible architectural backbone technologies. Out of a wide range of potential technologies Node-RED, Apache NiFi and Apache Camel are selected and compared. Those technologies with a diverse field of application are used to implement a comparable test setup. Furthermore, they are analysed through their characteristics of processing, execution, usability and simplicity. As measured, Node-RED, Apache Camel and Apache NiFi indicate stable and fast message processing, especially in the case of raising message throughput. Node-RED surprises with constant memory and CPU loads and seems to be an exciting option in rapid prototyping.**

***Keywords*–System Architecture; Stream Processing; Message Routing; Complex Event Processing; Renewable Energy; Smart Grid.**

I. Introduction

Local energy networks in modern residential areas are used by additional installations such as charging stations for electric vehicles, local energy production (e.g. with photovoltaic systems) and energy storage systems. The installations could operate independently and contribute. However, an increase in efficiency would be conceivable if those installations worked in a coordinated manner. An integration platform required for this could collect and process information from all installations and influence an entire system to operate an optimised overall setup. Our research project "Wohnungswirtschaftlich integrierte netzneutrale Elektromobilitat in Quartier und Region" (WINNER) [2] aims to integrate such systems and wants to use such a platform to optimise the local energy consumption.

Therefore, three main tasks have to be considered and brought together by our so-called WINNER DataLab (WDL). At first, we collect all the produced data and store them in a meaningful way. After that, potentials have to be found, e. g., correlating weather forecasts, electricity consumption, specific time information, and the usage characteristic of electric vehicles (EVs) to optimise external energy purchase for charging batteries. In the end, we have to optimise operation. So we could control and accumulate electric energy locally or supply it to the grid. Maybe it is superior or necessary to get energy from another grid operator, e. g., in case of too less output of the local energy production.

The facts as mentioned earlier imply an information flow managed by data streams. These must be routed and checked for mistakes. Beyond various data sources have to be integrated, like Representational State Transfer (REST) interfaces based on Hypertext Transfer Protocol (HTTP) or mail services, as well as other proprietary transport and communication protocols.

According to backbone technologies, we have to discuss the potentials of tools made for message routing and analysing within the WDL. We focus on event-based approaches and easy integration of external components. In the end, we ask for a tool that offers the possibility of routing messages and analysis of the contained data without dropping information. For the decision-making process, a unified prototype was realised in a selection of possible tools, compared and considered under load. This publication summarises our decision process. We want to explicitly outline, that our aim is not to benchmark the preselected tools in detail. We want to observe the memory usage development as well as the CPU consumption of the tools while processing an increasing amount of messages. So, we can get an idea of the behaviour of the message processing systems.

In Section II, related work about terms and projects related to our approach are discussed. Section III presents the level 0 view of our WDL, and the following Section IV lists the requirements we impose on this system. As a result of that, a short overview of possible tools is presented in Section V. Furthermore, three tools are used to implement and compare a uniform task in Section VI and analysed them in Section VII by using measurement values of latency, memory consumption and CPU load. Finally, we discuss the results in Section VIII.

II. Terms and Related Work

The WDL seems to be far away from traditional database management systems. The WDL should be usable as a platform for data scientists for mining knowledge as well as a platform to attach analyses for already known behaviours and relationships directly on data streams. Furthermore, the WDL should consume data from different kinds of systems as well as produce data to optimise the usage of those systems.

Connecting various types of applications by using their provided data and processes belongs to the term of enterprise application integration (EAI) [3, P. 3]. Within the area of application integration terms like message-oriented middleware (MOM) and service-oriented architecture (SOA), as well as enterprise service bus (ESB), describe how to challenge those use cases [4, S. 1][5]. As a traditional approach, MOM describes how to use asynchronous messages to decouple applications based on messaging systems [5]. SOA, on the other hand, represents an architectural concept where applications publish their precisely defined functionalities within reusable services [5]. Finally, ESB draws an open standard, which merges those ideas and defines a distributed architecture used to integrate applications. The architecture itself describes calls and distribution of messages between integrated applications [5].

Within the scope of EAI, the enterprise integration patterns (EIPs) are the base of tools to solve integration problems. The EIPs describe a set of reusable patterns without a particular technology reference. Base concepts within these patterns are the usage of "routing" and "messages" [6].

Beside the application integration itself, activity tracking, sensor networks and analysing of market data is a central topic within the so-called complex event processing (CEP). CEP describes a general term for methods, techniques and tools. CEP helps to process events while they happen [7, S. 163].

Bringing together EAI, MOM, SOA, ESB and CEP seem to be not distinctly possible. Currently, there are multiple terms to describe the problem of integration, routing, processing and analysing. The first one, Information Flow Processing, is described in [8]. This term focuses on event processing in combination with data management to "collect information produced by multiple, distributed sources, to process it in a timely way" [8]. Another term, streaming data system, focuses on processing data streams within "a non-hard real-time system that makes its data available at the moment a client application needs it" [9].

While EAI, MOM, SOA, ESB and CEP are concepts to assemble setups based on already known behaviours and relationships between messages (or events), data sciences utilise tools to mine knowledge based on already available data. This part within the WDL uses concepts from knowledge discovery in databases (KDD), which describes methods to statical analyses, applications within the field of artificial intelligence (AI), pattern recognition and machine learning [10, S. 3].

In addition to this classification of concepts within our field of application, related work targeting onto architectural drafts in data grids and smart grids can be used. Chervenak et al. [11], e. g., describes basic principles for designing data management architectures and Tierney et al. [12] introduce concepts how to monitor such grids. Furthermore, Appelrath et al. describe in [13] the process of developing an IT-architecture for smart grids as a result of a German research project, and Rusitschka et al. [14] present a computing model for managing real-time data streams of smart grids within the scope of the energy market. But, these approaches are not directly applicable to our use case. Either they are large-scale, or focusing on data storing and mining. However, they can be considered within our architecture, which has to fill the gap between smart grids, data storing, possibilities for data mining as well as non-hard real-time event processing.

Besides the mentioned, more general architectures for the internet of things (IoT) are discussed in [15] and [16]. They point out that a flexible layered architecture is necessary to connect many devices. As an example, classical three- and five-layer architectures are mentioned here, e. g., as used within [17] and [18], as well as middleware or SOA-based architectures. Besides this discussion about architectures, [19] demonstrates the integration of IoT devices within a service platform which uses the micro-service architecture for this approach, which can be understood as a specific approach for SOA [20, P. 9].

In addition to IoT, measurement systems for IoT can also be considered. They also have to integrate different end systems. Further, they have to record different measured values and provide interfaces for analyses and calculations. For this, approaches like SOA or event-driven architecture (EDA) can be taken up, as demonstrated in [21]. This approach uses SOA and EDA in combination with an ESB. Using the micro-service architecture can be seen in [22] and [23] as loosely coupled by using the EAI [3, P. 3]. They describe a reference architecture using micro-services for measurement systems, which connects required data adapters as well as calculation and storage services, one more time through an ESB.

III. Architectural Draft

The WDL is a platform to gather, analyse and provide information to optimise the operation of the WINNER-project setup. Until now, this platform seems to be a data streaming platform which has to solve various integration tasks. At this point, the level 0 view of the WDL is discussed. Taking a look at the data sources and data sinks help to get a better understanding of what the WDL should do.

At first, there are external services. They are sending messages to the WDL, or it acquires data from them. These data packages have to be assumed as heterogeneous, e. g., carsharing data of a booking system the WDL is connected to. That means the WDL gets information on bookings like start time and end time. Out of that, current state updates on a reservation such as an earlier beginning or a defect vehicle can be received. Another data source offers messages containing information on the current electrical power consumption. The actual electricity price is obtained by an interface of European Energy Exchange (EEX). Data of photovoltaic systems or batteries are gained through System, Mess- und Anlagentechnik (German solar energy equipment supplier) (SMA) interfaces. The German Meteorological Service [24] offers historical information on the past weather, an application programming interface (API) of the online service OpenWeatherMap [25] is available for weather forecasts.

A system working with time series, forecasts and master data must be created. As visualised in Figure 1, the WDL is positioned between the sources above, and at least five data sinks. These refer to controllable devices like a charging station, a battery or Smart Home systems. On the other hand,

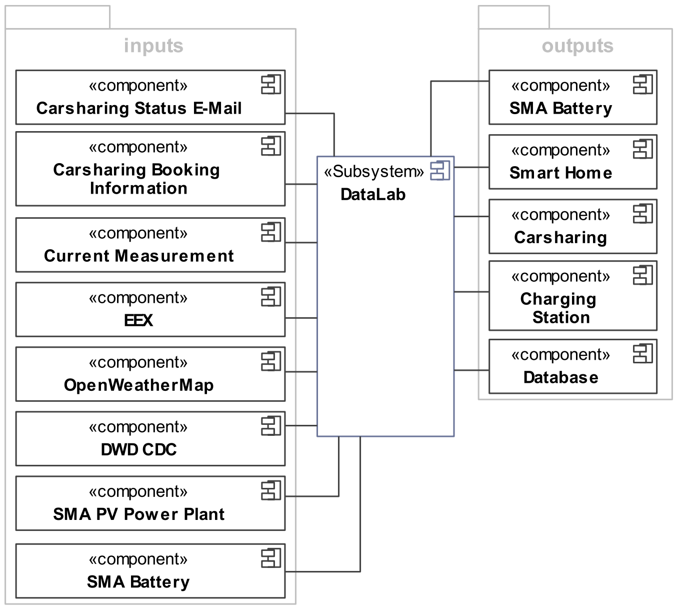

inputs
«component» Carsharing Status E-Mail
«component» Carsharing Booking Information
«component» Current Measurement
«component» EEX
«component» OpenWeatherMap
«component» DWD CDC
«component» SMA PV Power Plant
«component» SMA Battery
«Subsystem» DataLab
outputs
«component» SMA Battery
«component» Smart Home
«component» Carsharing
«component» Charging Station
«component» Database

Figure 1. Level 0 view of the WDL.

data is delivered to the carsharing service and dumped to a database. Within our setup, a KairosDB is used as our data storage interface as it is suitable for working with larger time series and quite easy to use in combination with a Cassandra backend for storage.

An unanswered question is how the different components can be integrated and how analysis, as well as event processing, can be handled. The WINNER project focuses on the integration of components of the residential area into the Smart Grid. That means predictions must be made to get an overview of the future power consumption and the electricity production. Either one charges the batteries or one uses the stored energy to overcome load peaks. The prediction mechanism might be implemented by using artificial neural networks (ANNs) or regression methods. Thinking about energy production predictions, it might be neccessary to receive information from hardware components like SMA-devices and weather services. These specific data formats require reshaping to use them in prediction mechanism. Using input filters, output filters, and stream routing the arriving information is transformed and sent to the prediction and dump units. These units save the data, send commands or just forward data to external devices.

IV. Requirements

One can divide up the list of requirements into three subsets. The first one refers to the system in general; the second touch the various components and the third covers the aspects of architecture and functional groups.

Thinking of the system in general shows that the ability to *process time series data* is required. An incoming message contains a time value referring to a point and a value, e. g., the result of a measurement. The WDLs task on an incoming message is to associate the arriving values with a data source. Possible data sources are photovoltaic installations, batteries, power consumption measurement devices or actual weather data. Out of that, the system must handle forecast data. They are unique because a complete time series and a time value, which refers to a validity point are included. At the time this point describes the time series is valid. Contemplable data sources are weather forecast services or EEX. The last category covers master data without time dependencies. Booking information or general data on devices and services belong to this group. This data may be very unstructured like text-only entries.

Focusing on technical aspects derived from our architectural draft in Section III, the WDL needs the ability to *process JSON, XML and CSV values*. Out of that proprietary formats have to be handled as well. Especially photovoltaic and smart metering installations tend to send production data in proprietary formats.

Central non-functional requirements are *scalability and reliability*. The latter refers to interfaces receiving data from external services and devices. On occurring errors incoming and shortly arrived messages should not get lost. Referring to the message rates previous estimations showed, that our system has to process about 30 messages per second on average. Thinking of peak times, we estimate a message occurrence higher than the average by a factor of 10. Especially, if external information-providing services use caches to buffer data and send them shortly afterwards or in case of network problems this may happen.

Keeping the interfaces in mind, one has to think of the *necessary contact points to other services* or the environment in general. The consumer interfaces of the WDL have to accept HTTP requests, especially while communicating with REST services. Similarly, FTP servers must be communicated with. The WDL must receive and process e-mails as well. Likewise, a file-based data transfer is needed. Finally, there are interfaces to external services using proprietary communication formats via TCP or UDP. The developed system has to enable the reception of messages sent by them. In contrast, the message producing components of the WDL primarily need to communicate via HTTP. Particularly the interface to a database can be made up of simple REST client services sending HTTP-based messages.

After paying attention to input and output components, the *internal processes of routing and filtering* shall be characterised. Asynchronous processing describes an essential requirement. Message queues or small buffer databases may decouple various components so they can work without waiting for each other to terminate. Furthermore, incoming messages caused by occurring events have to be converted into an internal format. To achieve this, the WDL can extend these data packages with additional information. However, after processing unneeded contents must be removed as well. Alongside external descriptors have to be mapped to internal descriptors and vice versa.

The WDL has to *transform the incoming data into an internal format* for further processing. Additionally, the WDL has to be capable of providing data for doing manual statistical evaluations and analysis. Furthermore, the WDL has to be capable of triggering automatic evaluations and forecasts as additional components. This work is done while keeping the CEP pattern in mind.

Within this paper, we leave out the specific aspect of data storage. That means different databases are not discussed or compared. The built prototype uses a KairosDB to persist

TABLE I. TOOL OVERVIEW AND CLASSIFICATION. CLASSIFICATION IS BASED ON TOOLS TO HANDLE EAI, CEP KDD AND RE.

Name	EAI	CEP	KDD	RE
Apache Camel	✓	✗	✗	✗
Apache Storm	✗	✓	✗	✗
Apache Spark	✗	✗	✓	✗
Apache Hadoop	✗	✗	✓	✗
Apache ServiceMix	✓	✓	✗	✓
Apache NiFi	✓	✗	✗	✗
Siddhi	✗	✓	✗	✗
ESPER	✗	✓	✗	✗
WSO2 CEP	✗	✓	✗	(✓)
RapidMiner	✗	✗	✓	✗
KNIME	✗	✗	✓	✗
Node-RED	✓	✗	✗	✗
JBoss Fuse	✓	✓	✗	✓
StreamLine	✗	✓	✗	✗
Hortonworks SAM	✓	✓	✗	✓

time series data. It was chosen because of an existing simple HTTP-based interface that provides easy access.

Furthermore, security and data protection with regard to the sensitive information collected is not taken into account in this analysis. The efforts required for this influence the resulting data throughput but are not relevant for this evaluation.

V. TOOL OVERVIEW

The WDL requirement analysis illustrates an EAI task with KDD topics. Furthermore, results gathered from KDD could result in CEP related tasks, which have to be considered as well. The following list of tools covers these tasks. Of course, this list is not complete. There are a lot of tools available to handle specific tasks within the area of EAI, KDD or CEP. Our selection focuses on widely used, platform independent and easily accessible tools with suitable licenses models. Thus, the list of our selection contains mainly open source tools.

Selected tools will be classified into at least one of our primary topics: (1) tools to handle KDD related tasks, (2) tools to solve EAI related tasks and (3) tools to implement CEP related tasks. Additionally, there are (4) tools providing runtime environments for executing solutions solved with tools from class (1), (2) and (3).

Table I lists our selection of considered tools. Furthermore, this table classifies them within our previously identified main topics. Apache Camel is an open source lightweight framework to solve EAI problems based on an implementation of EIPs in [6]. Furthermore, a lot of components are available to extend the functionality of Apache Camel [26]. Apache Storm is an "open source distributed realtime computation system" with a lot of use cases like "realtime analytics, online machine learning, continuous computation". This scalable environment can handle a lot of data streams within a specific Storm topology [27]. Apache Spark [28] and Apache Hadoop [29] are tools for knowledge discovery in data. They differ in performance as well as their internal approaches in data storage and processing. Apache Service Mix [30] and JBoss Fuse [31] are integration containers, which include other tools like Apache Camel. Apache NiFi [32] provides a web interface for configuring directed graphs, which represent the processing of incoming messages. It uses so-called flow files for information representation. Siddhi [33] and ESPER [34] are CEP engines and can be used as standalone tools as well as the integration of tools like Apache Camel. WSO2 CEP is a runtime environment for the CEP engine Siddhi, which adds user interfaces for external and internal usage [35]. RapidMiner [36] and KNIME [37] are tools for knowledge discovery in already existing data. It is also possible to integrate interfaces to access data streams and use a wide range of algorithms to analyse collected data. Finally, Node-RED is a message processing framework with IoT roots and can be used to solve application integration problems quickly. This framework is based on Node.js, can be extended with additional packages and deployed into cloud services like Bluemix [38]. StreamLine and Hortonworks Streaming Analytics Manager (SAM) are aimed to develop and deploy Streaming Analytics applications visually. Therefore, Hortonworks SAM and StreamLine provide bindings for different streaming engines, a rich set of streaming operators as well as operational lifecycle management [39].

VI. PROTOTYPE

According to [1] we have selected three tools based on our preselection in Section V and our experience. We want to use the tools within a uniform test setup. This selection focuses on tools from different fields of application: (1) Node-RED because of its simplicity within the field of IoT, (2) Apache NiFi because of its easily configurable data flow and (3) Apache Camel as the reference implementation for EIPs in combination with Wildfly as Java EE based runtime environment.

The comparison is done with a uniform test setup with a simplified task. This setup combines the integration of a REST-based data source which encodes data with JSON, a KairosDB based data sink with HTTP interface which consumes JSON encoded data as well, and a calculation of the mean according to a particular sender device, e. g., a photovoltaic station, has to be calculated across multiple messages. The time window of these multiple messages is ten seconds. That means if sender "Station A" sends a message at 10:00:00 am the values "Station A" sent between 09:59:50 am and 10:00:00 am are used for calculating the mean. The result is delivered via HTTP request to an external service which consumes JSON encoded data as well as the data source and KairosDB.

The data source in our test setup gets its messages from a generative photovoltaic data endpoint in configurable intervals. This source device transmits structured data like the tuple "(time, energy, station, id)". The first value of the generated data tuple represents a long value as a point in time, the second value a double based energy value of solar insolation. Furthermore, a tag containing the string based station name is included. Finally, the last value is a string based identifier of this single message for further time measurements. The identification value does not contain any relevant information in the context of energy data aggregation. It is only used to register and match the outgoing and incoming messages on the peripheral systems around the measurement environment.

The KairosDB endpoint of this setup gets its message as structured data like the tuple "(name, value, tags, time, id)".

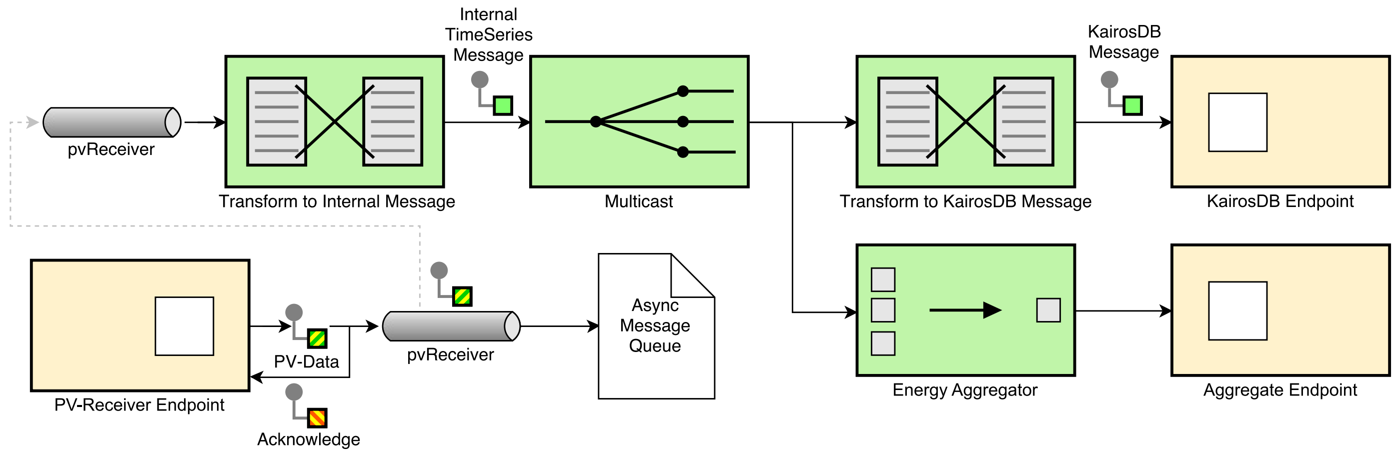


Figure 2. Visualisation of our general prototype based on EIP notation.

This tuple corresponds to the structure of data that are sent to a KairosDB instance for storing also. The included ID is not needed for the process of storing the data but necessary for matching the messages afterwards. Finally, the aggregation endpoint of this test setup gets its message as the same structured data like the data source tuple "(time, energy, station, id)".

The internal message routing has to be implemented across Node-RED, Apache NiFi and Apache Camel as shown in Figure 2. This figure illustrates the test setup and its components by using the notation of EIPs. First, there is an endpoint called "pvReceiver", which receives incoming messages and pushes them to a transformation step using an asynchronous message queue. Out of that, the endpoint ensures that the sender gets an acknowledge message. After transforming the information into an internal format, a multicast happens. So, one incoming message is sent to two other endpoints. These are the aggregate and the database endpoint. Before the data is sent, it is transformed to a matching database message format ("Transform to KairosDB Message" in Fig. 2) or the information is aggregated (("Energy Aggregator" in Fig. 2)).

The selected transformation and routing steps refer to the already mentioned requirements in Section IV and architectural draft in Section III to cover some data source, transformation, processing, reverse transformation as well as dumping.

A. Node-Red

Node-RED is a JavaScript-based message processing framework with IoT roots and can be used to solve application integration problems quickly. The framework is executed with Node.js and uses NPM for dependency management. Implementing the test setup mentioned above within Node-RED web client can be done by using a bunch of function nodes, nodes to create HTTP endpoints as well as change nodes. Change nodes are designed to modify the structure of our currently handled message object. Function nodes, on the other hand, are designed to execute custom scripts onto a particular message. Finally, there are nodes to create HTTP endpoints. Examples are HTTP server nodes to some path which can be called, HTTP response nodes which have to be placed within a message processing path which starts with an HTTP server node and HTTP client nodes to call external resources.

The implemented setup is shown in Figure 3. As mentioned, the messaging pipe starts with "PV Receiver" to create an HTTP server endpoint for "/endpoints/pvenergy". The message is piped onto an HTTP response node as well as to the primary processing path. The path starts with a function node to clean, enrich and transform incoming messages into the internal format. The result is forwarded to the database handling as well as the aggregation processing. Our database handling creates KairosDB compatible messages by using a template node and submits the resulting message by using an HTTP client node. The aggregation processing utilises the other function node to implement the aggregation function. This function node describes a simple memory to persist messages within a time window of ten seconds as well as calculating the mean within this window for the particular installation. The aggregation handling is finalised with a switch node to determine "NaN" values and an HTTP client node.

Summarising, Node-RED is a platform which is quickly providable for fast prototyping which can integrate various data sources as well as data sinks. But, it is tricky to develop collaboratively. Well, each developer can maintain its environment, but Node-RED itself manages Node-RED-Flows; synchronising them between different development platforms is hard. Furthermore, any particular use case, e. g., aggregating values from messages has to be implemented manually or by using additional NPM-based components which can be added directly in Node-RED. However, it is possible to integrate a broad range of endpoints with standardised formats and protocols. Handling proprietary endpoints requires more efforts in development.

B. Apache NiFi

Apache NiFi is a tool that runs within a Java Virtual Machine (JVM). A graphical user interface is offered within the web browser. Multiple of the so-called "Processors" can be used for standard tasks like receiving and sending HTTP requests ("PVReceiver" or "PostToKairos" in Fig. 4). Out of that, one can use custom processors by providing external JavaScript files (transform nodes in Fig. 4) or external java packages ("EnergyAggregator" in Fig. 4).

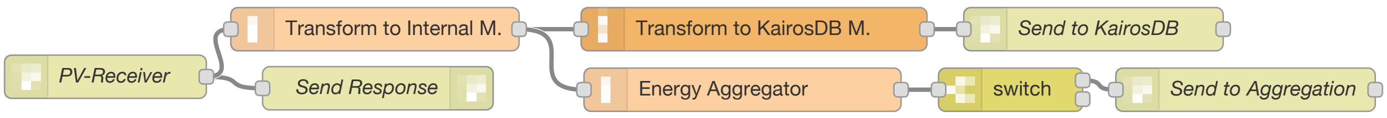


Figure 3. Node-RED implementation of the example from Section VI.

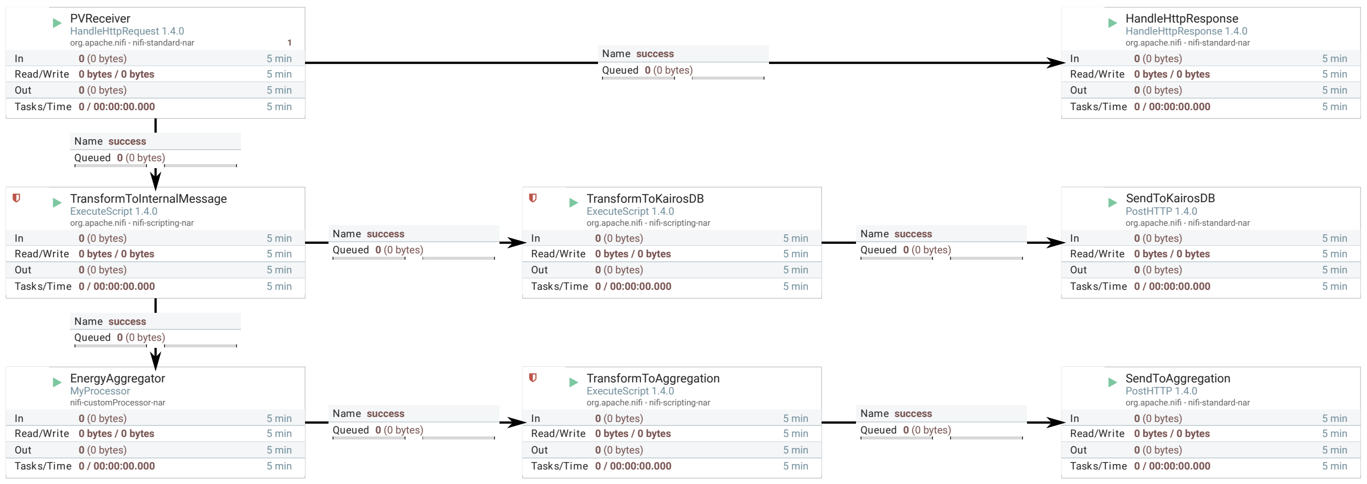


Figure 4. Message processing using Apache NiFi.

The "PVReceiver" accepts JSON-formatted data sent to the matching internet protocol (IP) address on port 8000. The processors "SendToKairos" send two different message formats to two different destinations ("http://nifireceiver:1880/dbmessages" and "http://nifireceiver:1880/averagemessages"). After receiving a message via POST request, it is answered by the "HandleHttpResponse" processor. The message is transformed into an internal format and processed on two different paths. The upper one (Fig. 4) transforms the message to another format and sends it to a Kairos endpoint. The lower branch aggregates the energy values of the messages in a way as mentioned above and sends them to the database as well. In contrast to our general setup (Fig. 2), we used an additional formatting step before publishing to the aggregation endpoint. So, we were able to separate the calculation from the formatting step.

We have to mention some application-specific facts. The incoming information is distributed within Apache NiFi by using so-called "Flowfiles". These contain attributes added by the processors like HTTP header data. Out of that, the user can add attributes to custom scripts. So, we use the attributes to map the energy and time values that should be processed within our use case. In the end, the "TransformToKairos" processors (Fig. 4) take the matching attribute values and put them into the outgoing message. The "EnergyAggregator" (Fig. 4) internally uses a map to calculate station-specific averages over the last ten seconds. Thus, a list of measurement values and timestamps is managed for each station. Old values are removed from the list, so each calculation happens on the actual values.

We configured the "PVReceiver" with an internal queue size of 1000 requests. Out of that, every queue between the processors is configured with a "maxWorkQueueSize" of 10^6 and a "maxWorkQueueDataSize" of 1 GB. If we had not done that, overfull queues would cause the preceding processors to pause their work. In theory, this "maxWorkQueueSize" enables Apache NiFi to keep all incoming messages within one single queue.

Finally, we can say, that Apache NiFi provides a nice workflow for creating custom processors and integrating own functions. JavaScript can be used via external script files. Furthermore, a Maven template can be used to create processors using Java. Every manipulation of the data flow graph within the web browser causes a history file, which is stored in an archive directory. So, a rollback can be done quickly. We want to outline, that there were no problems of dockerizing this application. It is easy to provide the script files and processor archives by using volumes and corresponding configuration files.

C. Apache Camel and Wildfly

Apache Camel is a Java-based EAI-framework, which is lightweight and extendable. It can be executed as a standalone routing system or within middleware infrastructures like Spring, Java EE, Apache ServiceMix or JBoss Fuse. Implementing the test setup mentioned above within Apache Camel can be done by utilising a REST endpoint and describing a route which channels incoming messages to our HTTP database and aggregation endpoints. Apache Camel offers a large number of implemented patterns, which are described in [6], as well as the option to implement custom processes, for example within "Beans". Furthermore, it is possible to extend the framework with own components for

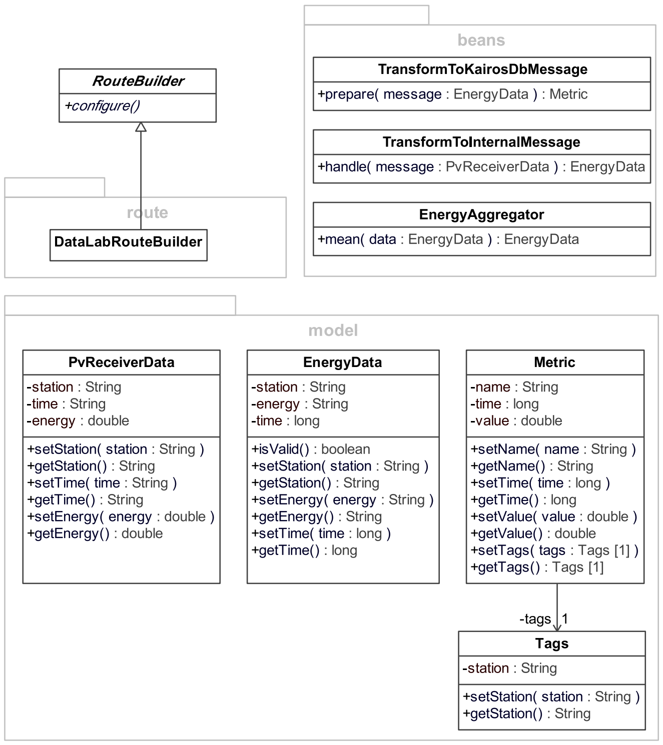


Figure 5. Apache Camel implementation of the example from Section VI.

further functionalities.

Fig. 2 visualises general and the finally implemented route within Apache Camel. Its components are shown in Fig. 5. The route itself is implemented by using the so-called “Java Domain Specific Language (Java DSL)” in Apache Camel. This route is implemented within “DataLabRouteBuilder” and describes the REST endpoint, which uses a servlet to process a specific resource and utilises SEDA to decouple incoming message flows from database and aggregation flows locally. SEDA is a lightweight in-memory message queue component within Apache Camel. The decoupled route contains the transformation and enrich bean “EnrichPvReceiverData” to transform external “PvReceiverData” into internal “EnergyData” as well as a multicast to handle the database and aggregation route. The database route contains another bean “KairosDbPrepare” to transform internal “EnergyData” into “Metric” datatypes for “KairosDb”. The aggregation route includes the aggregation bean “AggregationByInstallation” itself, which is implemented as stateful bean to save messages within a time window of ten seconds and finally calculate the mean for a particular installation. Both routes are completed with an HTTP client call onto the respective external endpoint.

Finally, Apache Camel is easy to use, primarily when used in combination with Maven as build and deployment tool. It is possible to describe routes within Java DSL, as we did, or use XML-based description to build those routes. Furthermore, Apache Camel is primarily a routing engine. Any particular use case, e. g., aggregating values over messages, has to be implemented manually or by using additional libraries.

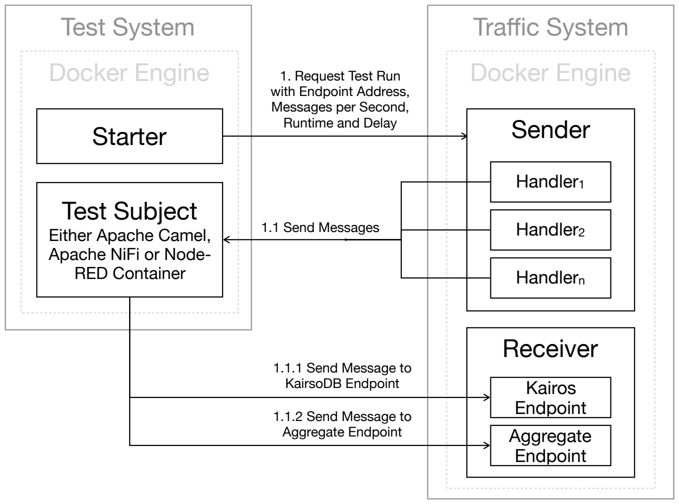

Test System
Docker Engine
Starter
Test Subject
Either Apache Camel, Apache NiFi or Node-RED Container
Traffic System
Docker Engine
Sender
Handler₁
Handler₂
Handlerₙ
Receiver
Kairos Endpoint
Aggregate Endpoint
1. Request Test Run with Endpoint Address, Messages per Second, Runtime and Delay
1.1 Send Messages
1.1.1 Send Message to KairsoDB Endpoint
1.1.2 Send Message to Aggregate Endpoint

Figure 6. Visualisation of the test setup and the resulting communication steps.

D. Comparison

Table II compares our three prototypes. The characteristics regard the type of development, how to configure the data flow, the programming language for development, available scripting languages, possibilities for extension, resulting artefacts and containerization abilities were compared. Differences between the tools and the prototypes result primarily in the type of development, which is either based on source code (Apache Camel) or a web-based GUI (Apache NiFi and Node-RED), as well as in the resulting artefacts. Apache Camel is packaged and run as a traditional JAR archive; thus, it can run in execution environments with installed Java. In the case of Apache NiFi and Node-RED, however, the deployment mainly revolves around the description of the processing (flow file) and the associated dependencies or any self-implemented extensions.

VII. Runtime Measurements

In this section, we want to test the prototypes mentioned above. Guaranteeing constant conditions for every application and run, Docker containers are executed on the same machine. These containers encapsulate the runtime environment as well as the prototype itself. Our test machine runs on Debian GNU/Linux 9.3 Stretch using an Intel(R) Core(TM) i5-4570 CPU @ 3.20GHz. We chose this machine because we target to reach the hardware limitations faster. Because of the main task of our prototypes is routing messages, some exclusions are necessary. First, the application sending information to the routing engine is installed on another machine. Furthermore, the service which receives information sent by the routing engine is placed on another machine too. This setup admits for quantifying the response time, memory consumption and CPU load of the various Docker containers or the applications within them omitting the aspect of additional load of sending and receiving applications.

The setup used for the test execution can be found under [40] as a source code repository and is visualised in Fig. 6. Within the repository, you can find various docker-compose files [41], which start the different tools and a so-called

TABLE II. Prototype comparison.

Characteristic	Apache Camel	Apache NiFi	Node-RED
Development	Source code	Web-based GUI	Web-based GUI
Configuration	Possible to include external configuration, default flow described and configurated in DSL with Java or XML	Embedded in flow, has to be edited by using GUI	Embedded in flow, has to be edited by using GUI
Programming Language	Java	Java	JavaScript
Scripting	-	Clojure, ECMAScript, Groovy, Lua, Python, Ruby	JavaScript
Extending	API available, has to deployed as JAR and included in classpath, also possible to use simple Java Beans without any API dependency	API available, deployed as JAR and has to be placed in classpath	API available, installed via NPM
Artifacts	Packed as JAR archive	Flow file and dependency JARs	Flow file and dependency list
Containerise	Common workflow with Maven and Docker, use Java runtime environment	Use Docker, add flow and dependency JARs to existing Apache NiFi container	Use Docker, add flow and dependency list to existing Node-RED container

"Starter Container". This one sends a message to the sender, transmitting the destination IP address of the tool container, a time delay and the duration of the sending process. Afterwards, the starter container is powered off. Using this approach, the tool to evaluate and the test configuration can take place on the test machine. The measurement process is started by it too, and a script automates the whole process.

The sending device transmits JSON-based structured data tuples like "(time, energy, station, id)". One could think of a solar power system with a particular station identifier sending the actual energy production. The frequency of transmitted messages is configurable and initially set to 200 per second. Later we increased the rate up to 800 messages. This procedure supplements the measurements of the formerly published paper [1]. The sending process itself had a duration of 600 seconds. After this sending time, we waited for 600 seconds for later arriving responses. So, our measurement phases took 1200 seconds overall.

The JVM for our Apache NiFi CEP instance and the Apache Camel prototype are fixed to 6 GB of space, which is fully allocated on startup. The "MaxPermSize" is set to 1 GB. We use the measured values of "jstat" for calculating the memory consumption of the tools mentioned above with a time resolution of one second. Furthermore, we sum up the usage of survival space ("S0U" and "S1U"), eden space ("EU"), old space ("OU"), metaspace space ("mu"), and compressed class space ("ccsu"). A node.js module measures the memory consumption of Node-RED, i. e., the "heapUsed" value. Out of that, the CPU load is measured by the "top" command every second. We get the response times of the various systems by measuring the time of sending and the time of receiving messages in milliseconds. The arrival timestamps of messages corresponding to database operations and aggregations are measured separately.

The following results are presented first in an error-specific manner. We refer to individual behaviour and found error cases. Afterwards, the tools are compared to each other.

A. Results

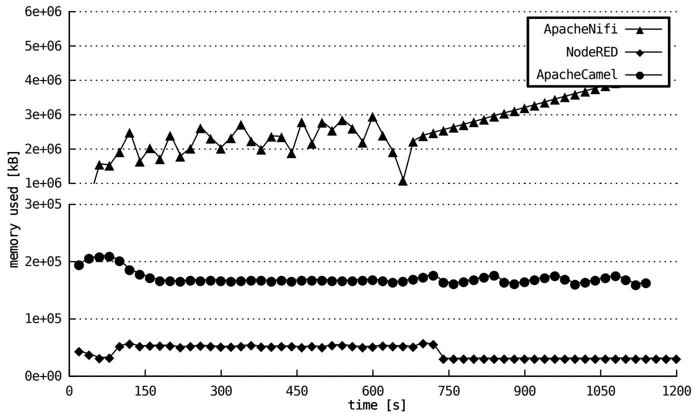
By sending 200 messages per second, we did not find any errors in message processing. Every tool did its part and processed all messages we sent. On higher message rates Node-RED did not act like the other tools. Beginning with a rate of 400 messages per second it did not answer the requests appropriately. About 602 of 239601 messages we tried to send were not answered, i. e., our sender delivered errors. The rate of 800 messages per second caused about 226,303 TCP-related errors, like "EADDRNOTAVAIL", "ECONNRESET" or "ETIMEDOUT" while 480,001 messages were sent overall. For this high message rate, we can state, that Node-RED was able to process only the half of the messages correctly. Apache NiFi shows another behaviour. Sending 800 messages to it, about 63,000 messages of 479,201 cannot be processed. In contrast to Node-RED, we got the HTTP error 503 for a not available service, i. e., the message could not be processed at all, database and aggregation message. Apache Camel always answered all messages send to it.

We were caused by the error messages to research more intensively on the mistakes happen. We can state that the TCP handshake for Node-RED did not happen in the right way. We checked the amount of opened ports of the involved Docker containers and the machines, but there is no lack. We state, that the Javascript event processing loop itself needs that much CPU and memory, that not all requests of the sender get handled quick enough. So, many messages the sender tries to dispatch are not delivered to the processing parts within Node-RED.

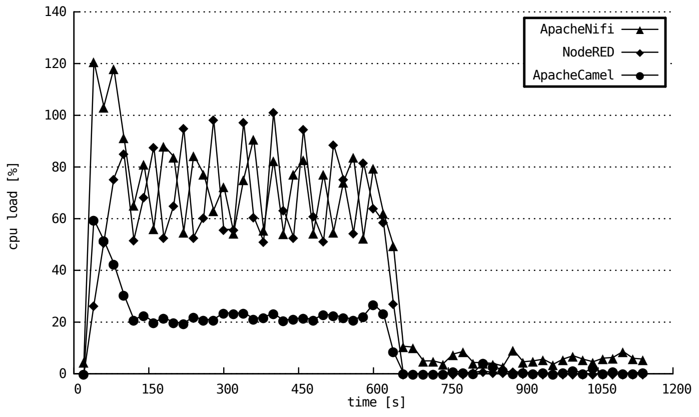
Summarizing we can say, that our measurements got difficult because of instabilities of the Docker Engine. Further analyses may use a more clean Docker setup, i. e., shutting down the service or restarting the test machine before each series of measures.

B. Comparison

The memory consumption (Fig. 7(a) and Fig. 9(a)) does not change much across all measurements (200 up to 800 messages per second). A factor of 10 is between the memory consumption of Apache Camel or Node-RED and Apache NiFi. The "S1U" value measured by "jstat" is the main cause of the higher RAM consumption of Apache NiFi. We want to mention the decreasing memory usage of Apache Camel

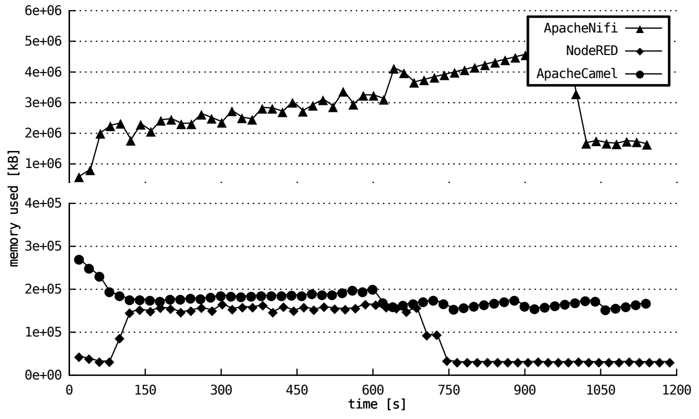


(a) Memory Consumption

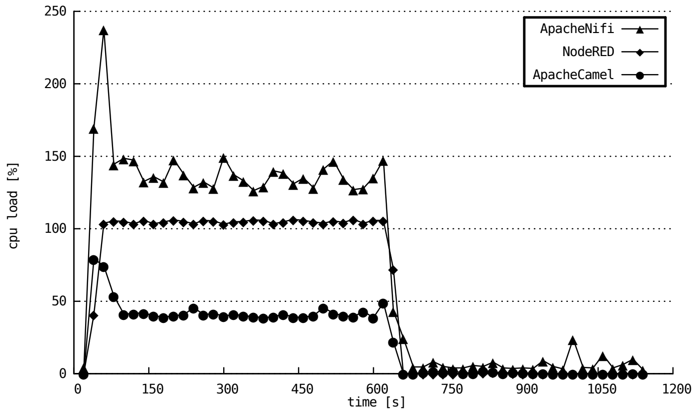


(b) CPU Usage

Figure 7. Memory consumption and CPU load while sending 200 messages per second (values aggregated over 20 s).

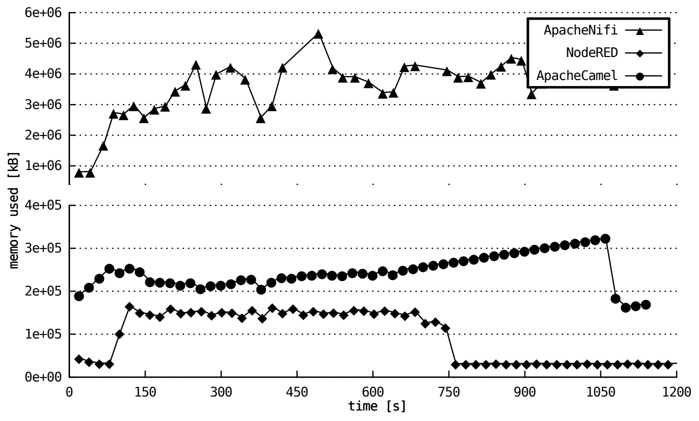


(a) Memory Consumption

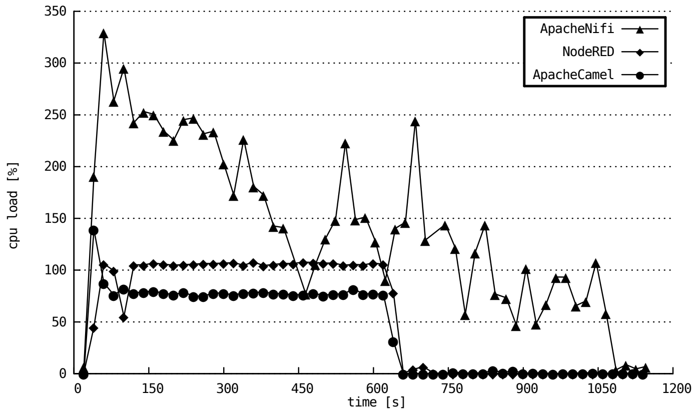


(b) CPU Usage

Figure 8. Memory consumption and CPU load while sending 400 messages per second (values aggregated over 20 s).

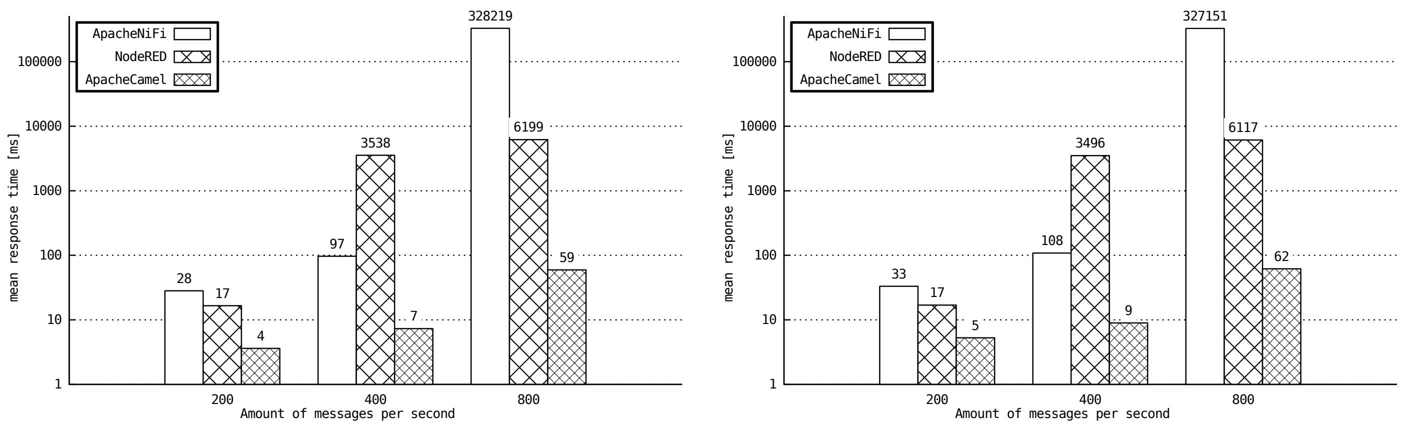


(a) Memory Consumption



(b) CPU Usage

Figure 9. Memory consumption and CPU load while sending 800 messages per second (values aggregated over 20 s).



(a) KairosDB Endpoint

(b) Aggregate Endpoint

Figure 10. Mean response times of tested systems with various message frequencies for database (a) and aggregation (b) messages.

while processing messages. According to the former published paper [1] we state, that "eden space utilisation" causes the less memory consumption while processing a more significant amount of messages. This finding is also present this time.

Watching the CPU graph (Fig. 9(b)) we can state, that Apache NiFi forces a much higher load than the other tools. Apache Camel showed an expectable process from a CPU usage of approximately 22 % for 200 messages per second up to 40 % for 400 messages per second and 80 % for the rate of 800 messages. In contrast, Node-RED reached the limit of a CPU usage of 100 % by processing 400 messages per second (Fig. 8(b)) and kept this behaviour on 800 messages. Apache NiFi showed the highest usages of the CPU. At this point, we want to mention, that the higher CPU usages (more than 100 %) in Fig. 7(b) and Fig. 9(b) are caused by the usage of Java. Apache Camel and Apache NiFi are able to use multiple threads so that the sum of CPU usages can sum up to more than 100 %. Node-RED, in contrast, is limited to one single thread, as we would expect for a Nodejs application.

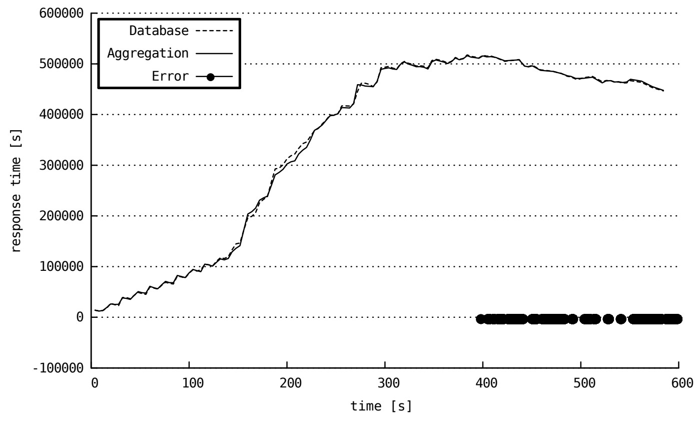
We researched more intensively on the behaviour of Apache NiFi. As visualised in Fig. 11(a), the response time increases rapidly after 150 seconds. The step pattern up to this point in time may cause by small garbage collecting processes, which clean the memory from processed message objects. At this time, Fig. 11(b) shows a nearly full available RAM. So, from this time Apache NiFi needs much time to process the incoming messages, because of a quiet full memory and the corresponding more comprehensive garbage collections. Out of that, shown in Fig. 11(a) on the lower right (plotted as negative values), the HTTP endpoint delivers errors to the sending component. The corresponding points in Fig. 11(b) show a drop in memory consumption. We state that Apache NiFi prevents its processors from overfilling the memory, even if not all messages can be received. Finally, it can see that the figures in Fig. 11 both correspond to each other. While we can see the ordinary memory behaviour of a java program on the right after 1080 seconds, we see the highest response times on the left at approximately 410 seconds. This timespan is approximately the time after the end of sending (600 seconds) when the memory consumption takes a regular course.

Other essential measurement values are the response times of the various tools, i. e., the time between sending a message and getting the calculated result for aggregation or the reformatted database message. As mentioned above (Section VII-A) not all messages were answered. We calculated the mean response times overall sent and received messages. So, Fig. 10 must be considered from the point of view that Apache NiFi was not able to process about one-eighth of the messages (63357 of 479201) at the highest rate. Out of that, Node-RED caused errors for the half of the messages (226734 of 480001) the sender tried to transmit. The results show quite similar times for database (Fig. 10(a)) and aggregation messages (Fig. 10(b)). However, especially while watching the logarithmic scale, we can state, that Apache Camel answers our requests very fast, even for high loads. Apache NiFi needs more time for high message rates, i. e., four orders of magnitude, but answers the requests in case of a not available service with a well-formed error code. Node-RED causes errors in the sending process and needs a higher answer time (two orders of magnitude) than Apache Camel.

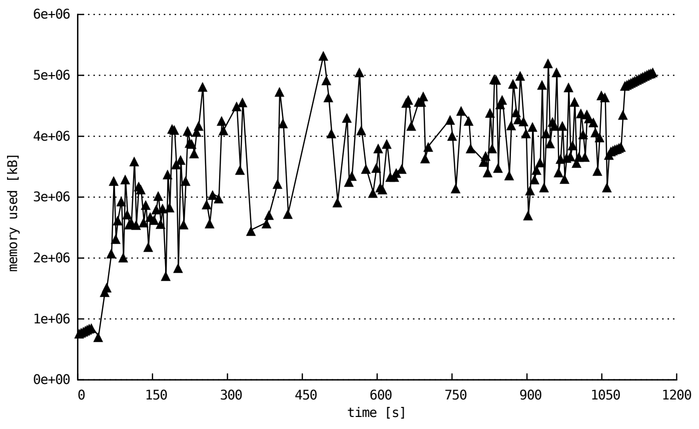
We tested Apache NiFi in an additional setup. The incoming messages had to be passed through without any manipulation. The tool only had to accept a request and send the message content to an HTTP interface. The measured response times were in the range of 9 milliseconds (200 messages per second) up to 390 milliseconds (800 messages per second) by processing every message correctly. Even without aggregating or reformatting any message, Apache NiFi is slightly slower than Apache Camel.

VIII. Discussion

Within this section, we do not discuss the measurements, but the process of measurement itself. Some peculiarities were referring to, e. g., the network connection. At first, we have to state some problems referring to our measurement setup. We used a common SoHo-router for our initial test setup. But, sending 800 messages per second caused network problems of the router itself, so a complete reboot was necessary. Afterwards, we changed our network setup to a common network switch. The sender and the test machine were configured with static IP addresses. We got a more stable network infrastructure in that way. Overcoming this problem,



(a) Response Time



(b) Memory Consumption

Figure 11. Response times of Apache NiFi along the sending process of 800 messages per second (a) (3200 values aggregated, i. e., four seconds) and the corresponding memory consumption of Apache NiFi (b) (values aggregated over 4 s).

we found, that our sending node did not work correctly for high message rates (800 messages per second), at least in combination with the Node-RED test setup. Not all messages we prepared for sending produced log entries. Missing TCP responses caused this, the sender did not get them. So, we restructured our sending process to use multiple threads and to produce more log entries, i. e., for creating a message, starting of sending process and logging errors. Referring to Apache NiFi, we may decrease its response time by using custom processors for message formatting instead of JavaScript using script executors. In total, to say is that a message rate of 800 ones per second forces our test setup. Especially Fig. 9(a) shows irregular time distances of the Apache NiFi data. This irregularity is caused by lags of the measurement tool itself, which is created using Java. Furthermore, the docker daemon on the test machine itself caused an unidentifiable error, which forced us to reboot the Docker Engine. We were not able to call commands within our test container or copy the locally created measurement files for memory and CPU usage. Our assumption referring to this is a broken docker process. For further measurements, we should think of regular reboots. Maybe a process controlled by Wake-On-LAN is more suitable for this use case because we would restart the entire machine instead of single processes, that can cause errors.

IX. Conclusion

The WDL has to be able to handle data streams as mentioned in different manners. Beside the integration and routing itself, there are tasks in the area of complex event processing as well as knowledge discovery in data. This is our second reflection of architectural backbone technologies covering those aspects. We tested high message rates forcing the test machine up to the limit of the hardware. Based on our experiences and measurements gathered from this test setup, we can make some decisions. In the case of a complex heterogeneous environment with different kinds of interfaces, Apache Camel seems to be a right choice. It is used within a wide range of conditions and able to handle many technologies to cover integration problems. Furthermore, this tool can manage high message rates by using reasonable memory.

Node-RED is a well-suited tool for rapid prototyping and IoT. On higher loads, it causes errors that are hard to handle. So, a conceivable approach could be inventing message processing using Node-RED at first. Afterwards, a more efficient implementation could be done using tools like Apache NiFi or Apache Camel. Node-RED might be usable as front-end system to easily integrate standardised external interfaces as well as an additional platform for experiments within a productive setup. Nevertheless, everything which can be done with Node-RED seems to be possible with Apache Camel too. The main difference can be found in the usability, the deployment process and the underlying language. Adapting knowledge discovery in such setups, independent of which routing engine is used, should be possible by using a database and route messages as required or by integrating available public interfaces from tools for knowledge discovery within Apache Camel or Node-RED. Apache NiFi shows a stable behaviour, even in case of high loads. If the processing of a message cannot be guaranteed, we get an HTTP error code and can try to send the information later. Furthermore, it seems to be quickly integrable with Apache ZooKeeper to run it within a cluster. Such a setup may be possible with Apache Camel or Node-RED.

Further research could be done on the possibility of using the considered tools within a clustered environment. This environment could overcome load peaks and increase the availability of the system. Additionally, topics regarding security and privacy should be taken into consideration.

X. Acknowledements

The research project WINNER is funded by the Federal Ministry for Economic Affairs and Energy of Germany under project number 01ME16002D.

References

[1] Sebastian Apel, Florian Hertrampf, and Steffen Späthe. Evaluation of architectural backbone technologies for winner datalab. In *Proceedings of the Sixth International Conference on Smart Cities, Systems, Devices and Technologies*, pages 35–43, Venice, June 2017.

[2] Chemnitzer Siedlungsgemeinschaft eG. WINNER-Projekt, 2017. Website http://www.winner-projekt.de; 2018-02-01.

[3] D.S. Linthicum. *Enterprise Application Integration*. Addison-Wesley information technology series. Addison-Wesley, 2000. ISBN 9780201615838.

[4] David Chappell. *Enterprise Service Bus*. O'Reilly, 2004.

[5] Falko Menge. Enterprise service bus. In *Free and Open Source Software Conference*, 2007.

[6] Gregor Holpe. Enterprise integration patterns. In *Proceedings of 9th Conference on Pattern Language of Programs*, September 2002.

[7] Michael Eckert and François Bry. Complex event processing (cep). *Informatik Spektrum*, 32(2):163–167, 2009.

[8] Gianpaolo Cugola and Alessandro Margara. Processing flows of information: From datastream to complex event processing. *ACM Computing Surveys*, 44(3):1–70, 2012.

[9] Andrew G. Psaltis. *Streaming Data - Understanding the real-time pipeline*. Manning Publications Co., 2017.

[10] Christian Gottermeier. Data mining: Modellierung, methodik und durchführung ausgewählter fallstudien mit dem sas enterprise miner. Diplomarbeit, Universität Heidelberg, 2003.

[11] Ann Chervenak, Ian Foster, Carl Kesselman, Charles Salisbury, and Steven Tuecke. The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets. *Journal of Network and Computer Applications*, 23(3):187 – 200, 2000. ISSN 1084-8045. doi: 10.1006/jnca.2000.0110.

[12] B. Tierney, R. Aydt, D. Gunter, W. Smith, M. Swany, V. Taylor, and R. Wolski. A grid monitoring architecture, 2002.

[13] Hans-Jürgen Appelrath, Petra Beenken, Ludger Bischofs, and Mathias Uslar. *IT-Architekturentwicklung im Smart Grid*. Springer Gabler, Heidelberg, Germany, 2012.

[14] S. Rusitschka, K. Eger, and C. Gerdes. Smart grid data cloud: A model for utilizing cloud computing in the smart grid domain. In *2010 First IEEE International Conference on Smart Grid Communications*, pages 483–488, Oct 2010. doi: 10.1109/SMARTGRID.2010.5622089.

[15] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash. Internet of things: A survey on enabling technologies, protocols, and applications. *IEEE Communications Surveys Tutorials*, 17(4):2347–2376, Fourthquarter 2015. ISSN 1553-877X. doi: 10.1109/COMST.2015.2444095.

[16] G. Choudhary and A. K. Jain. Internet of things: A survey on architecture, technologies, protocols and challenges. In *2016 International Conference on Recent Advances and Innovations in Engineering (ICRAIE)*, pages 1–8, Dec 2016. doi: 10.1109/ICRAIE.2016.7939537.

[17] Miao Wu, Ting-Jie Lu, Fei-Yang Ling, Jing Sun, and Hui-Ying Du. Research on the architecture of internet of things. In *2010 3rd International Conference on Advanced Computer Theory and Engineering(ICACTE)*, volume 5, pages V5–484–V5–487, Aug 2010. doi: 10.1109/ICACTE.2010.5579493.

[18] Zhihong Yang, Yingzhao Yue, Yu Yang, Yufeng Peng, Xiaobo Wang, and Wenji Liu. Study and application on the architecture and key technologies for iot. In *2011 International Conference on Multimedia Technology*, pages 747–751, July 2011. doi: 10.1109/ICMT.2011.6002149.

[19] A. Krylovskiy, M. Jahn, and E. Patti. Designing a smart city internet of things platform with microservice architecture. In *2015 3rd International Conference on Future Internet of Things and Cloud*, pages 25–30, Aug 2015. doi: 10.1109/FiCloud.2015.55.

[20] Sam Newman. *Building Microservices*. O'Reilly, 2015.

[21] Jean-Louis Maréchaux. Combining service-oriented architecture and event-driven architecture using an enterprise service bus. *IBM Developer Works*, pages 1269–1275, 2006.

[22] M. Vianden, H. Lichter, and A. Steffens. Towards a maintainable federalist enterprise measurement infrastructure. In *2013 Joint Conference of the 23rd International Workshop on Software Measurement and the 8th International Conference on Software Process and Product Measurement*, pages 63–70, Oct 2013. doi: 10.1109/IWSM-Mensura.2013.20.

[23] M. Vianden, H. Lichter, and A. Steffens. Experience on a microservice-based reference architecture for measurement systems. In *2014 21st Asia-Pacific Software Engineering Conference*, volume 1, pages 183–190, Dec 2014. doi: 10.1109/APSEC.2014.37.

[24] Climate Data Center. Website http://www.dwd.de/DE/klimaumwelt/cdc/cdc_node.html; 2017-01-06.

[25] OpenWeatherMap, 2017. Website http://openweathermap.org/price; 2017-01-06.

[26] Apache Camel, 2015. Website http://camel.apache.org; 2016-06-22.

[27] Apache Storm, 2015. Website http://storm.apache.org; 2017-02-28.

[28] Apache Spark, 2015. Website http://spark.apache.org; 2017-02-28.

[29] Apache Hadoop, 2014. Website http://hadoop.apache.org; 2017-02-28.

[30] Apache ServiceMix, 2011. Website http://servicemix.apache.org; 2017-02-28.

[31] Jboss Fuse, 2016. Website https://developers.redhat.com/products/fuse/overview/; 2017-02-28.

[32] Apache NiFi, 2018. Website https://nifi.apache.org/; 2018-02-19.

[33] Siddhi Complex Event Processing Engine, 2017. Website https://github.com/wso2/siddhi; 2017-03-05.

[34] Esper, 2016. Website http://www.espertech.com/esper/; 2017-03-01.

[35] WSO2, 2017. Website http://wso2.com/products/complex-event-processor/; 2017-01-06.

[36] RapidMiner, 2017. Website https://rapidminer.com/; 2017-03-05.

[37] KNIME, 2017. Website https://www.knime.org; 2017-03-05.

[38] Node-RED, 2017. Website https://nodered.org; 2017-01-06.

[39] StreamLine - Streaming Analytics, 2017. Website https://github.com/hortonworks/streamline/; 2018-02-28.

[40] Architectural backbone evaluation source code, 2018. Website https://github.com/winner-potential/smart-2017; 2018-02-28.

[41] Docker, 2017. Website https://www.docker.com/; 2017-12-05.