UNIVERSITY OF TARTU

FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

Institute of Computer Science

Computer Science Curriculum

Lembit Gerz

An Akka-based middleware for Laboratory Information Systems

Bachelor's Thesis (9 ECTS)

Supervisor: Naved Ahmed, Ph.D.

Tartu 2015

An Akka-based middleware for Laboratory Information Systems

Abstract

Data exchange with medical analyzers is an important requirement of any laboratory information system. Analyzers and the information system exchange data such as analysis results, orders and patient information. In high-throughput laboratories, clinical tests are performed around the clock, therefore manual data entry can become a bottleneck in their usual workflow. In order to increase the effectiveness of the lab staff and reduce the chance of human error, the data exchange has to be automated.

Due to technical restrictions and the large number of communication protocols used by different analyzers, middleware software is often used to mediate communication between the information system and analyzers. An interruption in the data exchange can cause financial damage or even risk to life. Therefore, certain requirements should be considered during the implementation of such software to make it reliable and maintainable.

This thesis gives an insight into the laboratory middleware field and builds a middleware prototype using the Akka toolkit. As a part of the prototype, an interface between a hypothetical analyzer and a laboratory information system is also implemented. The implemented prototype is used as a reference to analyze up to which extent Akka is suitable for implementing middleware software that satisfies requirements such as fault-tolerance, concurrency and code reusability.

As a result, the prototype can be used as a minimal framework to connect additional analyzers with laboratory information systems.

Keywords

Middleware, actor model, Akka, laboratory information system, concurrency, fault-tolerance, component reusability, monitoring.

Akka teegil põhinev laboriinfosüsteemi vahevara

Lühikokkuvõte

Andmevahetus meditsiiniliste analüsaatoritega on oluline osa igast laboriinfosüsteemist. Analüsaatorid ja infosüsteem vahetavad andmeid nagu näiteks analüüside tulemused, tellimused ja patsiendi informatsioon. Suure koormusega laborites teostatakse analüüse ööpäevaringselt, mistõttutu võib käsitsi andmete sisestamine pudelikaelaks labori töös. Et suurendada laboripersonali efektiivsust ning inimeksimusi, tuleb andmevahetus automatiseerida.

Tehniliste piirangute ja andmevahetusprotokollide rohkuse tõttu kasutatakse sageli vahevara tarkvara, mis vahendab andmevahetust laboriinfosüsteemi ja analüsaatorite vahel. Katkestus andmevahetuses võib põhjustada rahalist kahju või isegi ohtu elule. Seega tuleb vahevara arendamisel arvesse võtta teatud nõudeid, et see oleks usaldusväärne ja hallatav.

See bakalaureusetöö annab ülevaate laboriinfosüsteemi ja analüsaatorite liidestamisest ning selle käigus valmib Akka teegil baseeruv vahevara prototüüp. Prototüübi ühe osana implementeeritakse ka liidestus ühe hüpoteetilise analüsaatori ja laboriinfosüsteemi vahel. Prototüübi näitel analüüsitakse, kuivõrd on Akka teek sobilik implementeerimaks vahevara, mis rahuldab nõudeid nagu veataluvus, paralleelsus ning koodi taaskasutatavus.

Tulemusena valmivat prototüüpi on võimalik kasutada raamistikuna, millel toetudes saab implementeerida liidestusi erinevate analüsaatorite ja laboriinfosüsteemide jaoks.

Võtmesõnad

Vahevara, aktorimudel, Akka, laboriinfosüsteem, paralleelsus, veataluvus, komponentide taaskasutatavus, monitoorimine.

Table of Contents

Index of figures

Index of listings

Chapter 1: Introduction

Every laboratory that performs clinical testing on a larger scale uses a Laboratory Information System (LIS). An LIS is software that provides the functionality to manage data related to clinical testing, e.g., the entry of orders and results, specimen processing and patient information [1]. The LIS is usually a web application that is used throughout the laboratory by the personnel. Another important part of the LIS is the ability to integrate with other systems, such as the national health record, a hospital information system or medical analyzers.

Medical analyzers are physical devices that reside in the laboratory and are used by the laboratory personnel to perform clinical analyses on specimens. For example, a hematology analyzer can measure the white blood cell count on a blood specimen.

Analyzers need to communicate with the LIS to perform the analyses. This includes, for example, information about the required analyses that should be performed on a specimen, information about the patient and the analysis results. While smaller laboratories might enter the data manually, this approach is not sustainable for larger laboratories and must be automated. Most analyzers provide an interface for exchanging data with other systems. This can be, for example, over a TCP[1] connection or a file-based protocol.

Some LIS vendors choose to connect the analyzers to their systems directly, including all the logic needed for the data exchange as a part of their LIS installation. However, this approach has some drawbacks [2]:

Technical restrictions. Some analyzers only support an interface that cannot be directly connected to a web-based LIS. For example, older analyzers can only communicate via a local filesystem. In that case the device must be connected to a host computer which is a regular PC that resides near the analyzer. The analyzer then uses the computer's file system to exchange messages (e.g., writes analysis results as files in a folder 'C:/results' and reads orders from files 'C:/orders'). This prevents connecting these analyzers directly to the LIS and requires the use of an intermediary that performs file-based IO and exchanges the data with the LIS over a web-based transport protocol (e.g., TCP or HTTP).

Lacking modularity. Analyzers employ a myriad of different data formats. These specify how the messages exchanged between the analyzer and LIS are structured (e.g., the patient name is located in the 5th field of the 3rd line in the message). There are some recommended

[1] Transmission Control Protocol

standards such as HL7[2] and ASTM[3] that many analyzer vendors support, but some still use their own custom data formats [2]. When the logic of processing all the different data formats is included in the LIS installation, the modularity of the system suffers. Consider the case when the laboratory needs to connect a new analyzer. After a new interface for that analyzer is implemented as a part of the LIS, the new version has to be deployed. A redeployment usually comes with temporary downtime. The work of the whole laboratory is disturbed because of one analyzer. It can be argued that the processing logic of different data formats is conceptually not a core part of an LIS. An analyzer and an LIS can be implemented as a client-server architecture, where the LIS (i.e., server) should know as little as possible about analyzers (i.e., clients).

To address these problems, *middleware* software is often used. In the context of this thesis, middleware is a piece of software that mediates all communications between the LIS and the analyzers. It translates different analyzer protocols for sending messages to the LIS and vice versa. That allows LIS vendors to expose a uniform interface to communicate with their system (e.g., a REST API) while the middleware takes care off all the nuances of different analyzer protocols. This also enables the middleware to be deployed separately without affecting the LIS [2].

The middleware can also apply additional control logic as it is mediating communication, such as filtering out certain messages, persisting the messages in a database and converting measurement units [1]. Figure 1 illustrates how the middleware plays a part in the data exchange. There are multiple analyzers connected to one instance of a middleware. The part of the middleware that serves an analyzer is called a *driver* for that respective analyzer.

[2] Health Level 7 - http://www.hl7.org/
[3] ASTM standards committee - http://www.astm.org/

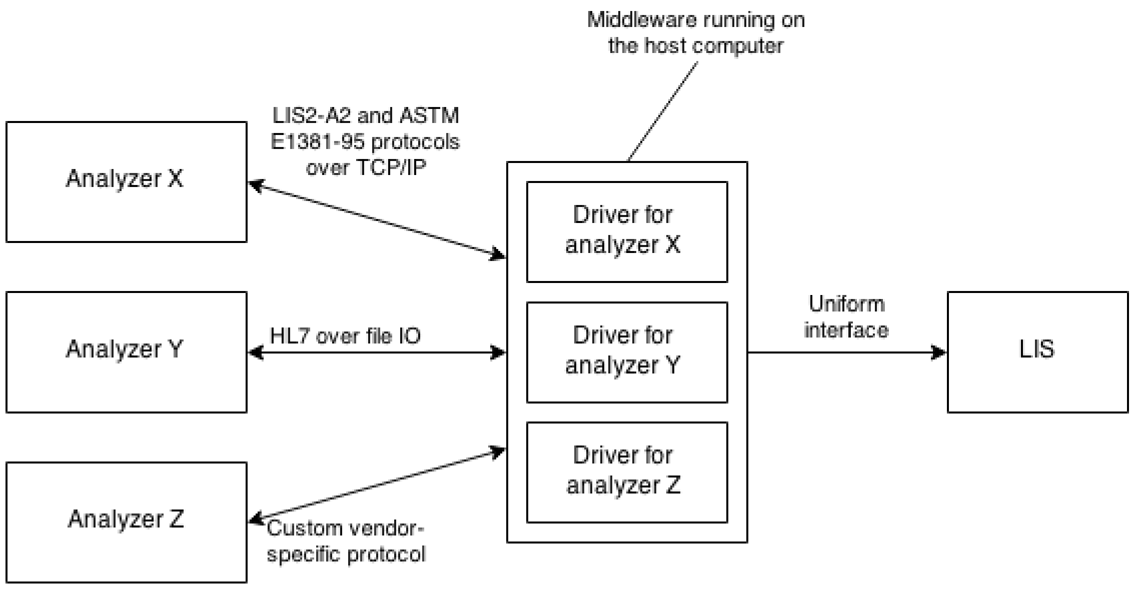


Figure 1: Conceptual model of the middleware and its interactions.

1.1. Problem description

This thesis proposes an implementation of a laboratory middleware using the Akka toolkit. The solution aims to provide a framework of the middleware that can be relied upon to add new drivers for other analyzers. As a reference implementation, a driver is implemented for connecting a hypothetical analyzer and an LIS.

The following criteria are established to evaluate the suitability of the proposed implementation and the Akka toolkit. These characteristics are important in a middleware application [1, 2]:

1. Parallelization and concurrency. A single installation of the middleware must be able to serve multiple analyzers without interfering with each other. For example, if there are two analyzers connected to the middleware, then the drivers for both analyzers should run in parallel.

In addition, processes within one driver should not be subject to concurrency-related problems, such as race conditions or deadlocks. This is a problem with bidirectional communication. For example, consider a situation when an analyzer is in the process of sending a message to the LIS. At the same time a message arrives from the LIS that should be delivered to the analyzer. If not handled correctly, this can result in a state where the middleware attempts to send a message to the analyzer when the latter is not ready to receive it. A correct implementation should finish receiving the current message and put any other messages from the LIS on hold until the analyzer has finished sending the current message.

2. Fault tolerance. If an error occurs during the execution of a specific driver, it should not affect other drivers. Also, the failure to process a message should not result in termination of the driver. Instead, the message that caused the error should be logged and the driver should continue processing the next messages.

3. Code reusability. Analyzer interfaces can usually be divided into two parts:

- **Delivery mechanism.** This specifies *how* the messages are transmitted to the analyzers and the LIS (e.g., file-based, HTTP requests or over TCP).
- **Data format.** This specifies the *format* of the message content. It outlines how certain data can be extracted from a message (e.g., the patient name is in the 4th field of the 2nd line).

Analyzers use a combination of these parts to communicate with the LIS. If a part for one analyzer is already implemented, then it is not beneficial to implement the whole communication protocol for another analyzer. For example, if there exists a driver that communicates HL7 over files, then implementing a driver that communicates HL7 over TCP should mean only writing code for the TCP delivery.

4. Monitoring. Finding the source of an error in retrospect is important as reproducing errors sometimes requires testing with the actual analyzer, making it expensive and time-consuming. Therefore, the message processing should be logged with enough thoroughness, so that the logs provide an accurate feedback about at which processing stage the error occurred. The middleware should effectively monitor the communications to quickly detect problems as soon as they occur.

5. Testability. Testing on real analyzers is often complicated, because it requires the cooperation of multiple parties (e.g., the developer, lab specialist and lab IT staff). Making changes to drivers that are used for life-critical analyzers is risky. Reusing code introduces the need for ensuring that after making changes to a reused functionality, all dependent drivers remain fully functional. The middleware implementation should provide a way to write automated tests that verify the correctness of the implemented analyzer drivers.

1.2. Thesis structure

The thesis is divided into 5 chapters. The first chapter gives a brief overview of the laboratory middleware domain and describes the problem addressed by this thesis. The second chapter explains the necessary background – the actor model and Akka, related work in the field and an example analyzer-LIS interfacing scenario. The third chapter discusses the implementation of the proposed middleware. The fourth chapter analyzes how the implementation satisfies the requirements raised in the first chapter. Lastly, a concluding chapter with summarizing notes and possible directions for future work. The appendix contains instructions on how to use files included in the extras that are submitted with the thesis.

Chapter 2: Background

This chapter starts by introducing the concept of the actor model and its Akka implementation. Next, provides an overview of related studies in the field of LIS middleware. Finally, an example scenario of an analyzer-LIS communication is illustrated. The scenario is used in Chapter 3 showing the implementation for solving the identified problem.

2.1. The Actor model

The actor model is a mathematical theory of computation that treats actors as the fundamental units of computation [3, 4]. An actor is a unit of computation within a set of actors called an actor system. The model is based on one-way asynchronous communication. An actor communicates by sending *immutable* messages to other actors. Once a message has been sent, it becomes the responsibility of the receiver. The sender can continue its work without waiting for a response. The message is placed in the recipient's mailbox and the recipient will process the message at a later time. When processing a message an actor can only do the following:

- send messages to other actors,
- create more actors,
- specify how it handles the next message it receives (i.e., change its internal state).

Erlang[4] was the first language to prove that the actor model could be used in large scale mission-critical software systems. Erlang was developed for telephony applications and was used at Ericsson to develop a product with over 1.7 million lines of code which is also considered as one of the most reliable of the company's products [5].

[4] http://www.erlang.org/

2.2. Akka

Akka is an implementation of the actor model for the Java Virtual Machine (JVM). Akka allows to combine actors with the object oriented programming paradigm, whereas the actor model suggests treating everything as actors. In addition to providing an API to write applications following the actor model, Akka provides many built-in features and recommended practices that ease the development process. The Akka toolkit is implemented in Scala, but also provides an API for the Java language. Akka can be taken into use by including a single *jar*-file into the Java project.

2.2.1. Actors

In Akka, an actor is a container for the following [6]:

- **State** – data exclusive to the actor that cannot be directly accessed by other actors.
- **Behavior** – the definition of actions to be taken on receiving a message.
- **Mailbox** – the buffer for messages that are sent by other actors.
- **Children** – the set of actors that have been created by the actor. Actors in Akka form a hierarchy – when an actor creates another actor, it automatically becomes its parent.
- **Supervision strategy** – the parent of an actor is known as its supervisor that must handle the failures of its children. The supervision strategy defines the action to be taken when a child actor fails.

Listing 1 shows an example of how an actor can be defined using the Java API for Akka. An actor in Akka is a subclass of *AbstractActor* and can have an *internal state*. In its constructor, the actor defines how it handles the messages it receives depending on the message type (*Message type 1-3*). All actors in Akka are hidden behind an *ActorRef* object, which acts as the address for the actor. It exposes a limited API, most importantly the *tell* method, which can be used to send messages to the actor. This ensures that the internal state of the actor cannot be accessed or modified by anything but the actor itself. The only way to communicate with the actor is to send it one of the types of messages it is able to process. An actor's interface to other actors are its address (e.g., *ActorRef*) and the types of messages it can process.

```
public class Greeter extends AbstractActor {

  private final List<Greet> greetHistory = new ArrayList<>(); // internal state

  private Greeter() {
    receive(ReceiveBuilder
      .match(Greet.class, msg -> { // Message type 1
        greetHistory.add(msg);
        System.out.println("Hello, " + msg.greetee);
      })
      .match(SayFarewell.class, msg -> { // Message type 2
        System.out.println("Goodbye, " + msg.farewelssssslee);
      })
      .match(Stop.class, __ -> { // Message type 3
        System.out.println("Stopping");
        context().stop(self());
      })
      .build());
  }
}

public static void main(String[] args) {
  ActorSystem system = ActorSystem.create();
  ActorRef greeter = system.actorOf(Props.create(Greeter.class)); // ActorRef
  greeter.tell(new Greet("Peter"), ActorRef.noSender()); // tell method
  greeter.tell(new SayFarewell("Peter"), ActorRef.noSender());
  greeter.tell(new Stop(), ActorRef.noSender());
}
```

Listing 1: Akka actor implementation using the Akka API for Java.

2.2.2. Actor systems and supervision

Actors in Akka are part of an actor system and form a tree-like hierarchy. Each actor has one parent actor that created it (also known as *supervisor*). Figure 2 shows an example of an actor system. In this figure, actors `/`, `user` and `system` are top-level actors that are created automatically by Akka for internal purposes. Actors `A-F` are user-created actors and form a hierarchy (e.g., `B` is a child of `A` and the parent of `E`, `F`).

The hierarchical structure of the actor system is useful for two reasons:

- **Delegation**. A good practice in actor systems is to split tasks into smaller subtasks. For example, if actor `B` is responsible for searching a keyword, it can split the search space between `E` and `F`. When `E` and `F` have finished, `B` aggregates the results. This helps in both reasoning about the code and allows to execute some parts of the tasks in parallel.
- **Supervision**. When a child actor fails (i.e., throws an exception) it suspends itself and its children and notifies its supervisor (i.e., parent actor) about the failure. The supervisor has four directives to choose from for handling the failure:
    - Resume the child actor (i.e., continue processing the next message).

- Restart the child actor, clearing out its internal state.
- Stop the child actor permanently.
- Escalate the failure to its own parent.

Actors are the units of fault-isolation, which is key to designing fault-tolerant software [5]. When an actor fails and is restarted or resumed, it does not affect other actors in the system. Other actors holding an address to the actor can still send messages to its mailbox. Restarting the actor does not clear its mailbox, so no messages sent to it are skipped, except the message that caused an error.

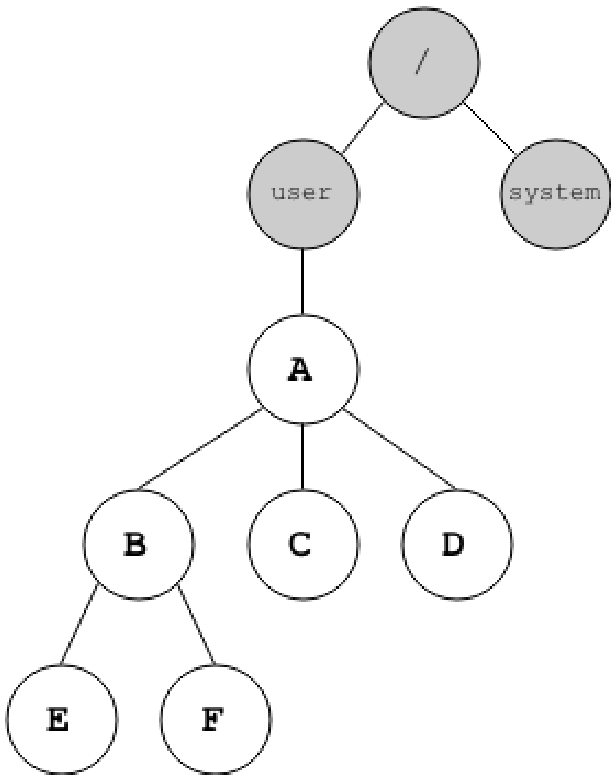


Figure 2: Actor system hierarchy.

2.2.3. Difference between conventional multithreading

Concurrency

One of the main aspects that makes concurrent programming difficult is shared state [7]. When two threads need to communicate, they must do so by mutating a shared piece of memory. For example, message-passing between two threads can be implemented by initializing the threads with a shared message queue – one thread adds items into the queue while the other periodically checks the queue for new messages. Mutating shared state requires explicit synchronization (e.g., locks). Incorrect synchronization can result in race conditions, deadlocks or corrupt state [7].

The actor model takes a different approach that removes the need for explicit synchronization. All mutable state inside an actor system is encapsulated within actors, which can mutate it while processing a received message. Since messages are immutable and message processing within an actor is single-threaded, there cannot be multiple processes accessing the same state at a given time. Isolating mutable state and enforcing immutable messages guarantees implicit synchronization and makes the actor model inherently parallel [7].

Fault-tolerance

Writing fault-tolerant code in Akka also differs from a thread-based approach. When an exception is thrown in the execution of a thread and it is not handled, the thread stops and there is no way to restart it other than instantiating a new thread. The common approach is to adopt a style of defensive programming to minimize the risk of a thread crashing [5]. This is the practice of trying to validate all the input data and handling all the possible errors that can occur with the aim of not letting an unhandled exception stop the thread. With this approach, code readability suffers as business logic becomes intertwined with the error-handling code. A programmer reading the code afterwards must figure out the intent of the author – which errors can actually happen and which error-handling code is added “just in case” [5].

Error-handling in Akka follows the “let it crash” philosophy known from Erlang [5]. This approach discourages defensive programming and advises that actors are fail-fast. It means that an actor should only do as it is supposed, otherwise it should fail and let its parent handle the failure. Since actors are the units of fault-isolation, the failing actor can be restarted by the parent without affecting the rest of the system. This also allows to write more concise code that clearly communicates the intent of the author.

2.3. Related work

Most of the existing LIS middleware solutions currently available are commercial products that are not freely obtainable. These solutions serve well for large LISs, but are not feasible for smaller laboratories. This section presents the existing freely obtainable solutions and related research in the LIS middleware field.

2.3.1. Mirth Connect

Mirth Connect is an open-source healthcare interface engine [8]. It is designed to be a generic tool to connect different medical information systems (e.g., LIS, hospital information system or national health record). The commercial version also includes some functionality for connecting medical analyzers. It features a graphical user interface for configuring different channels between a source and a destination system. The interface can be used to configure the processing logic for the exchanged messages, such as filtering, transformation, data extraction and routing. For more complex use cases, additional processing logic written in JavaScript can be added. Though the free version has no built-in support for analyzer integration, it is open-source and therefore can be modified for interfacing an external analyzer.

Mirth Connect is quite feature-rich and general to facilitate a variety of use cases, but this has the downside of complexity and a large application size of nearly 140MB. This reduces the portability of the application as deploying and updating it on multiple computers can be slow.

2.3.2. Froid - Free Open Instrument Middleware

Froid is a proposed solution for an open-source LIS middleware [10]. It is managed by Bika-LIMS, which is an open-source laboratory information system. Although the development is in early planning phase, the objectives of Froid are promising as it aims to be a vendor-neutral instrument server that can be used with any LIS as an interface to any analyzer. Another goal is to establish a community around Froid to encourage collaborators to share their implementations of analyzer drivers.

2.3.3. LabTechie

LabTechie is middleware that was developed for interfacing analyzers in two different laboratories in USA [9]. It is designed to interface analyzers that employ a file-based communication protocol using comma- or tab-delimited text files. The middleware parses the files' contents and converts them into XML format. These XML messages can be sent to the LIS and are also stored in a database for archival purposes. XSLT and XPath are used for extracting data and converting the messages into different formats. A graphical user interface can be used to search previously exchanged messages, manage the connected analyzers and manually verify the results before they are sent to the LIS.

LabTechie is not publicly available and the implementation details are not discussed. Although the middleware is stated to be general-purpose, modifying it for other communication protocols and LISs is not discussed. Another possible downside of LabTechie is that is requires Oracle's XML Database, which includes licencing fees.

2.4. Example scenario

To better illustrate the task that the middleware must fulfil, a concrete example is presented below. Figure 3 illustrates the workflow between the analyzer and the LIS.

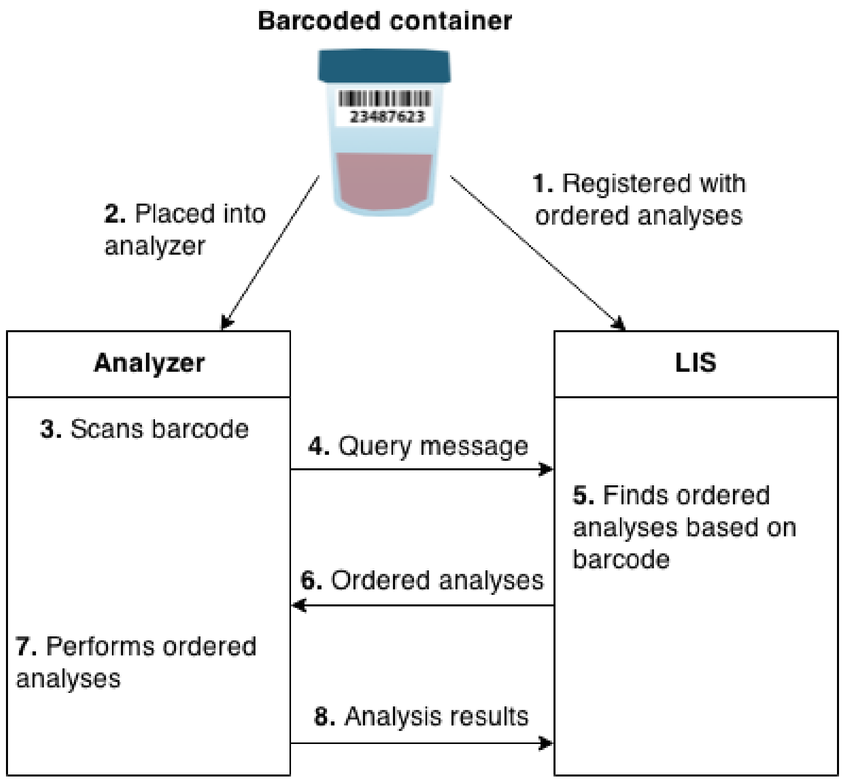
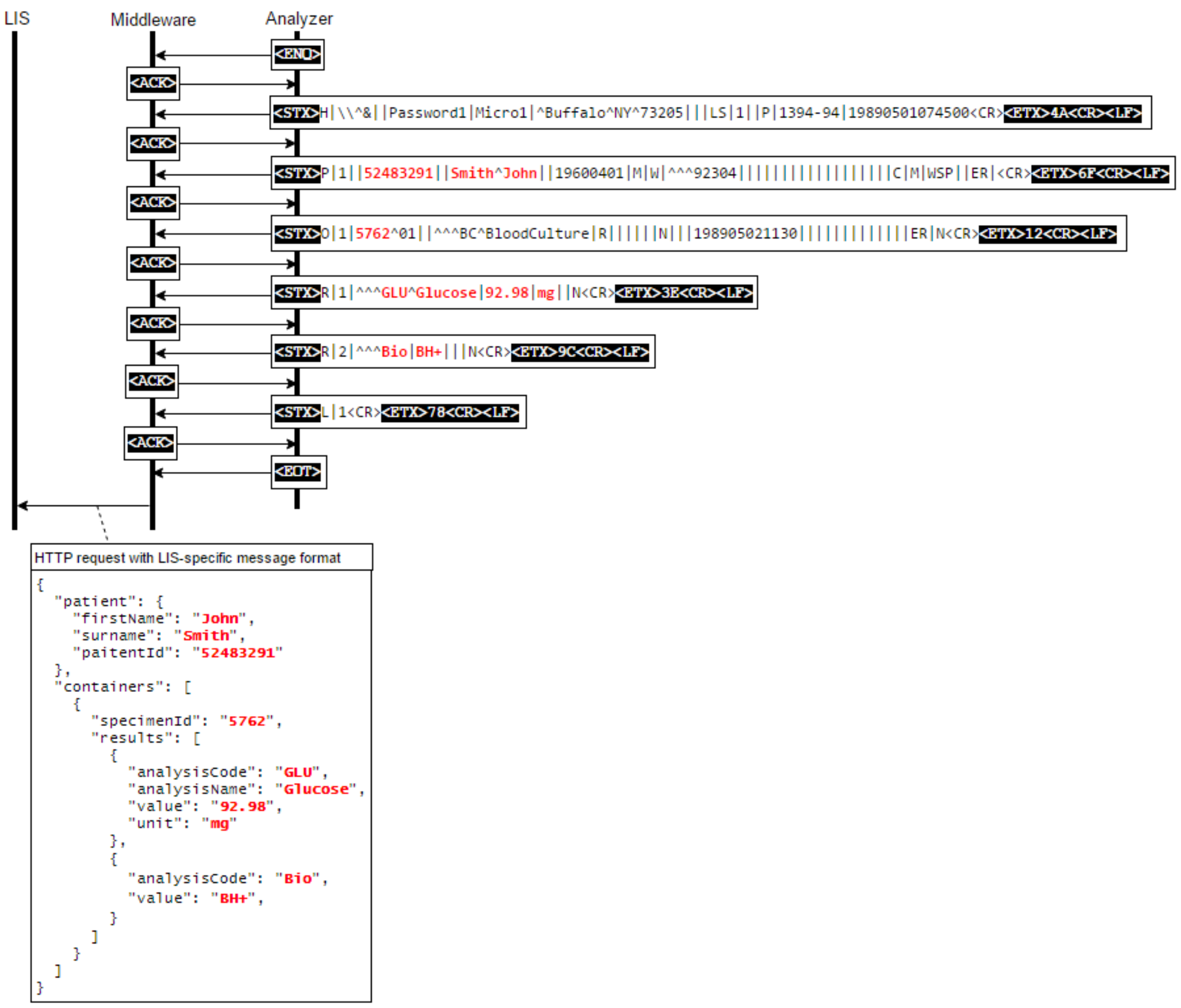


Figure 3: Conceptual overview of an analyzer-LIS message exchange.

A container with a specimen (e.g., blood or urine) is registered in the LIS and given a barcode (also called a specimen ID). The set of analyses that should be performed on the specimen is also registered in the LIS (**1**). The barcoded container is placed into an analyzer which has a barcode scanner (**2**). After scanning the barcode (**3**), the analyzer queries the LIS to determine which analyses it should perform on that sample (**4**). The LIS finds the previously registered container based on the barcode in the query (**5**) and sends the set of ordered analyses to the analyzer (**6**). After performing the analyses (**7**), the analyzer sends the analysis results to the LIS (**8**).

The analyzer communicates over a TCP connection. The data is transmitted following the ASTM E1381-95 low-level protocol [11]. This specifies sending the message line by line with the receiving party responding with an acknowledgement after each line. The data format of the message content follows the LIS2-A2 specification [12]. This is a higher level protocol, which specifies the message structure (i.e. where in the message certain data can be found). The LIS accepts and responds to HTTP requests using a custom JSON message format. The middleware stands between the analyzer and LIS mediating the data exchange by translating the analyzer protocol into the LIS protocol and vice versa.

Figure 4 illustrates the data exchange sequence when the analyzer sends an analysis result message. The message parts with the black background make up the abovementioned low-level protocol, the text with a white background enclosed between the low-level characters is part of the high-level protocol. The analyzer starts communication with an enquiry message (`<ENQ>`). The middleware signals its readiness with an acknowledgement (`<ACK>`). When all the lines of the message are transmitted, the analyzer sends the end-of-transmission message (`<EOT>`). The middleware then aggregates the received lines into one message, converts the message into the required JSON format and sends it to the LIS via an HTTP request.



```
<STX>H|\\^&||Password1|Micro1|^Buffalo^NY^73205|||LS|1||P|1394-94|19890501074500<CR><ETX>4A<CR><LF>
<STX>P|1||52483291||Smith^John||19600401|M|W|^^^92304||||||||||||||||||C|M|WSP||ER|<CR><ETX>6F<CR><LF>
<STX>O|1|5762^01||^^^BC^BloodCulture|R|||||N|||198905021130|||||||||||||ER|N<CR><ETX>12<CR><LF>
<STX>R|1|^^^GLU^Glucose|92.98|mg||N<CR><ETX>3E<CR><LF>
<STX>R|2|^^^Bio|BH+|||N<CR><ETX>9C<CR><LF>
<STX>L|1<CR><ETX>78<CR><LF>
```

```
{
  "patient": {
    "firstName": "John",
    "surname": "Smith",
    "paitentId": "52483291"
  },
  "containers": [
    {
      "specimenId": "5762",
      "results": [
        {
          "analysisCode": "GLU",
          "analysisName": "Glucose",
          "value": "92.98",
          "unit": "mg"
        },
        {
          "analysisCode": "Bio",
          "value": "BH+",
        }
      ]
    }
  ]
}
```

Figure 4: Message exchange sequence between an analyzer, middleware and LIS.

In addition to the minimal middleware framework, a reference driver for this scenario is implemented as a part of this thesis. Next, Chapter 3 illustrates a detailed implementation of the proposed middleware using the scenario introduced in this section. The hypothecial LIS in this example is named MyLab.

Chapter 3: Implementation

This chapter provides an overview of the implementation details of the proposed middleware prototype.

3.1. General architecture

The application is divided into two separate modules that can be deployed separately:

- **Back-end**. This is a Java application which includes the core of the message processing logic.
- **User interface**. This module is used to monitor the state of the back-end module. It is a web application implemented with TypeScript[5].

3.2. Back-end module

The back-end module includes the core of the message processing logic. It is a single actor system making up the actor hierarchy shown in Figure 5.

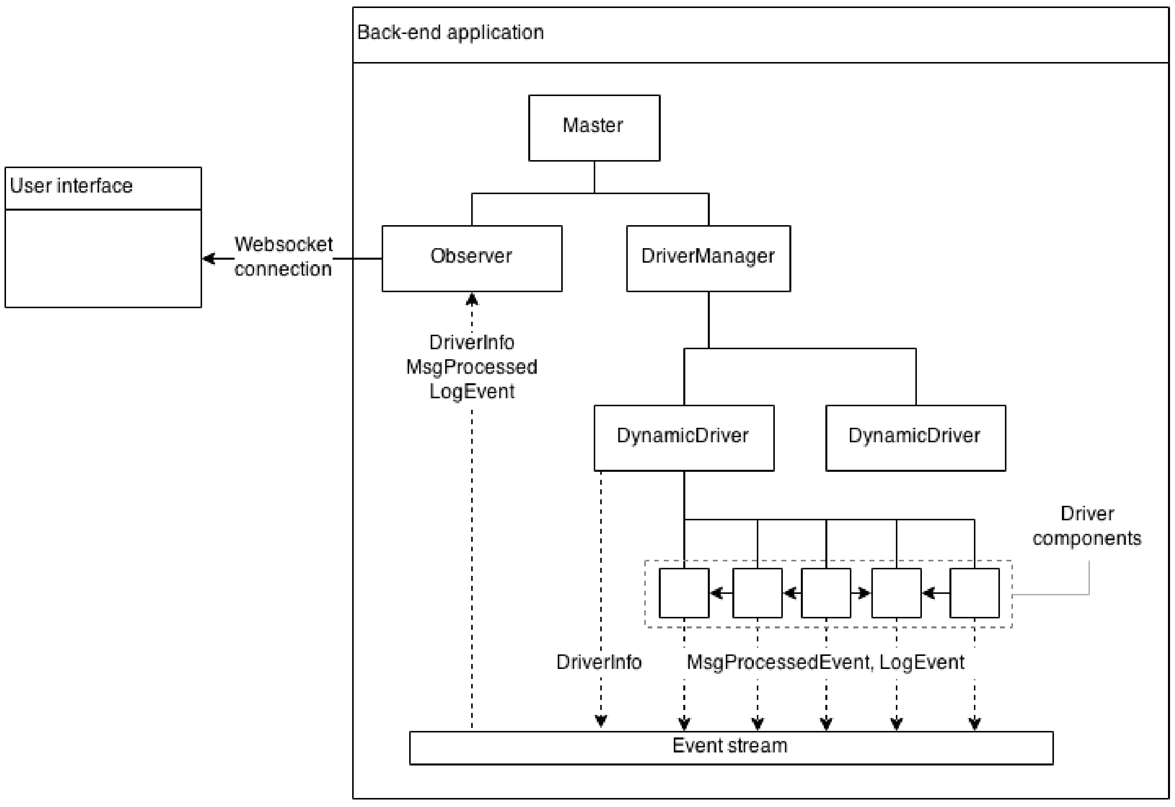


Figure 5: Back-end module architecture.

[5] http://www.typescriptlang.org/

Nodes in the hierarchy represent actors which have the following roles:

- ***Master***. The top-level actor which loads the configuration and starts the *Observer* and *DriverManager* actors.
- ***DriverManager***. This actor's task is to start the necessary drivers depending on the configuration passed to it by the *Master* actor (the two *DynamicDriver* actors in Figure 4).
- **Any number of *DynamicDriver* actors**. The *DynamicDriver* actor initializes and supervises the driver components that are needed to communicate with a specific analyzer. There is one *DynamicDriver* actor for every analyzer connected to the middleware instance.
- ***Driver components***. A set of cooperating actors that mediate the data exchange between the LIS and a specific analyzer (see Section 3.2.1.).
- ***Event stream***. The *Event stream* is a feature provided by Akka. It is a publish-subscribe bus that can be accessed by all actors in the actor system. Actors can publish messages to the *Event stream* and subscribe to receive messages published by other actors.
- ***Observer***. This actor collects information from the *Event stream* about the state of the drivers and messages that they exchange. The collected information is forwarded to the user interface.

3.2.1. Driver architecture

The *Driver components* shown in Figure 4 are actors that are directly responsible for the data mediation between an analyzer and the LIS. The structure of the components is based on the Pipe and Filter Pattern [13]. This pattern suggests dividing the message processing into isolated components and arranging them into a pipeline. Each component performs a single step of the processing logic and passes the result to the next component in the pipeline. This is analogous to the notion of "piping" in a Unix-based command-line.

Consider the example introduced in Section 2.4., where the analyzer sends an order inquiry to the hypothetical LIS called MyLab asking to retrieve the set of ordered analyses. As a response, the analyzer receives an order message containing the set of analyses that need to be performed on a given sample. Figure 5 shows how this exchange can be separated into components (this is a detailed view of the „Driver components" section in Figure 4).

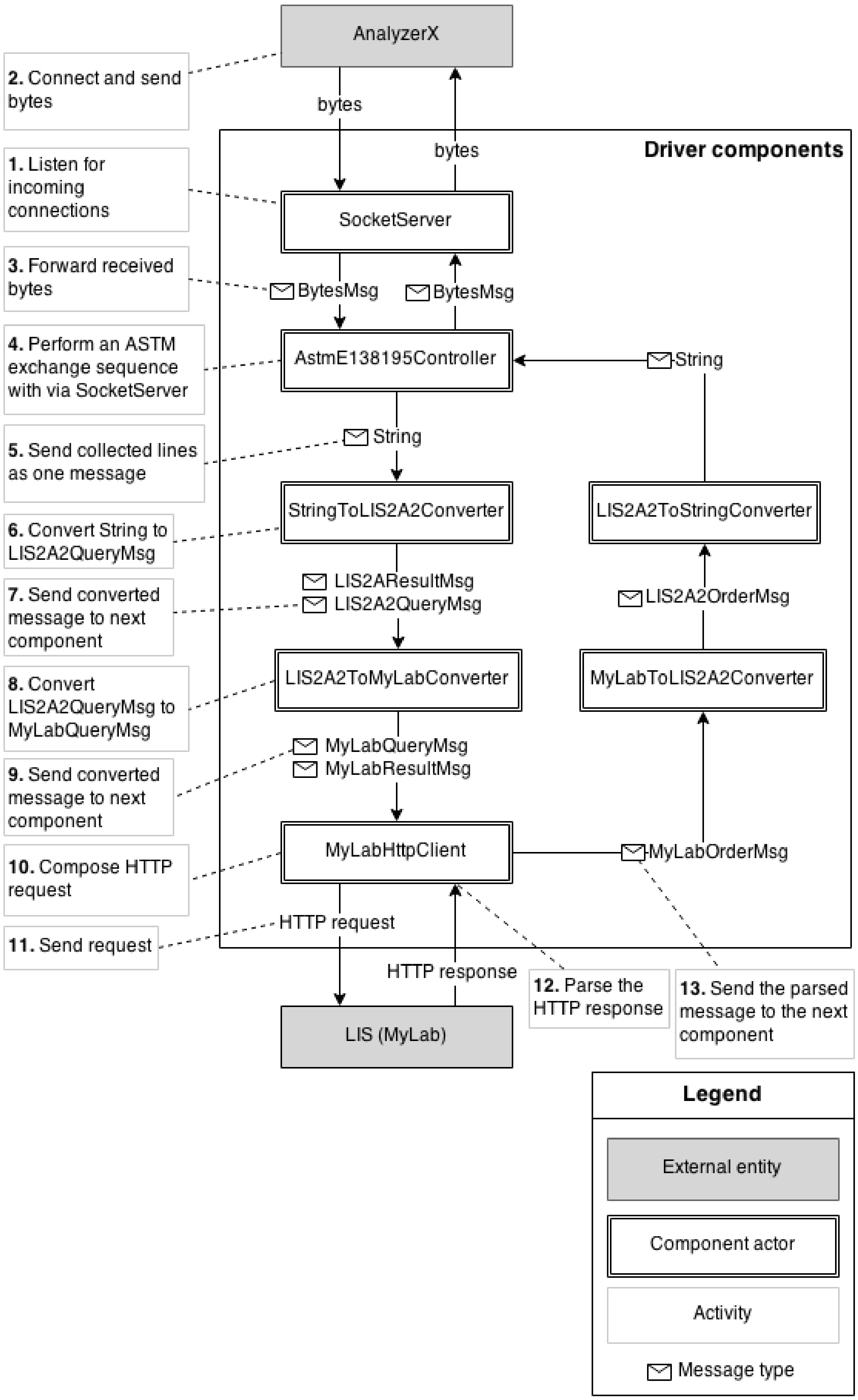

AnalyzerX
2. Connect and send bytes
bytes
bytes
Driver components
1. Listen for incoming connections
SocketServer
3. Forward received bytes
BytesMsg
BytesMsg
4. Perform an ASTM exchange sequence with via SocketServer
AstmE138195Controller
String
5. Send collected lines as one message
String
6. Convert String to LIS2A2QueryMsg
StringToLIS2A2Converter
LIS2A2ToStringConverter
LIS2AResultMsg
LIS2A2QueryMsg
7. Send converted message to next component
LIS2A2OrderMsg
LIS2A2ToMyLabConverter
MyLabToLIS2A2Converter
8. Convert LIS2A2QueryMsg to MyLabQueryMsg
MyLabQueryMsg
MyLabResultMsg
9. Send converted message to next component
MyLabHttpClient
MyLabOrderMsg
10. Compose HTTP request
HTTP request
11. Send request
HTTP response
12. Parse the HTTP response
13. Send the parsed message to the next component
LIS (MyLab)
Legend
External entity
Component actor
Activity
Message type

Figure 5: Driver architecture.

The *AnalyzerX* and *MyLab* nodes are not part of the driver and represent the actual analyzer and LIS that communicate via the middleware. Nodes inside the *Driver components* area are the components which mediate the data exchange between *MyLab* and *AnalyzerX*. Arrows that connect the nodes show the direction of messages that are exchanged between the components. An arrow is annotated with the type of the message a particular component passes to the receiving component.

The *SocketServer* starts a TCP server (**1**) that enables *AnalyzerX* to establish a connection (**2**). *SocketServer* reads the byte stream sent by the analyzer and forwards it to *AstmE138195Controller* (**3**). *AstmE1381Controller* performs the ASTM E1381-95 exchange sequence by responding through the SocketServer with an acknowledgement message after each line of the message (**4**). After receiving the end-of-transmission message, *AstmE138195Controller* aggregates the received lines into a string and sends it to *StringToLIS2A2Converter* (**5**). *StringToLIS2A2Converter* parses the received string into *LIS2A2QueryMsg* (**6**), which is a data structure that can be easily queried for individual parts of information. The parsed message is sent to *LIS2A2ToMyLabConverter* (**7**). *LIS2A2ToMyLabConverter* extracts the necessary data from the received message to construct a *MyLabQueryMsg*, which the LIS is able to process (**8**). This message is sent to *MyLabHttpClient* (**9**). *MyLabHttpClient* constructs the HTTP request with the JSON representation of the received message as the request body (**10**) and sends the request to MyLab (**11**). The HTTP response contains the order in the MyLab-specific format. The response is parsed into a *MyLabOrderMsg* (**12**) and forwarded to *MyLabToLIS2A2Converter* (**13**).

The rest of the message processing is carried out analogously to the previous steps. The order message is transformed into the analyzer-specific format and forwarded via *AstmE138195Controller* and *SocketServer*.

A driver component is an actor that performs a single part of the overall message processing logic. For each actor and message type, there is a corresponding Java class (e.g. *SocketServer.java* and *LIS2A2OrderMsg.java*).

Each component actor extends the class *FlowComponent*, which intercepts every message before it is processed. It publishes the following information to the *Event Stream* for every intercepted message:

- receiving component (i.e., current component),

- sender component,
- message content,
- processing start and end times,
- the stack trace of the exception, in case an error occurred while processing the message.

The published messages are collected by the *Observer* actor that forwards them to the user interface module.

3.2.2. Driver configuration

Each component is initialized by the *DynamicDriver* actor with a configuration object, which contains the information needed for the component to operate (e.g., the TCP address and port for the *SocketServer* and the MyLab URL for the *MyLabHttpClient*). Each configuration object also contains the address for the actor that the result of the current processing step should be sent to (e.g. *SocketServer* has the address to *AstmE138195Controller*).

At source-code level, the components of a driver are independent actors, which are not bound to each other at compile-time. The information about which components are started with a driver and how messages flow through them is specified via runtime configuration.

Listing 2 shows the configuration files for the sample driver implementation. File */driverDefinitions/LIS2A2OverTCP.conf* contains a general configuration for any analyzer that communicates LIS2-A2 over TCP. It specifies the components that are a part of the driver and how they are linked. As more drivers are implemented, the corresponding general configuration files can be added to the */driverDefinitions* folder.

File */application.conf* contains the configuration for a specific middleware instance. The *DriverManager* configuration entry specifies the drivers that should be started to serve the analyzers connected to the middleware instance. The example in Figure 2 imports the *LIS2A2OverTCP.conf* driver configuration and sets the analyzer specific configuration parameters to serve the *AnalyzerX* analyzer.

/application.conf

```
Drivers = {
 LIS2A2OverTCP = {
  include "driverDefinitions/LIS2A2OverTCP.conf"
 }
//  add an entry for every driver, e.g:
//  HL7OverFile = {
//    include "driverDefinitions/HL7OverFile.conf"
//  }
//  ...
}

DriverManager {
 AnalyzerX = ${Drivers.LIS2A2OverTCP} {
  SocketServer {
   address = "127.0.0.1"
   port = 50000
  }
  MyLabHttpClient {
   resultUrl = "http://localhost:8070/result"
   queryUrl = "http://localhost:8070/query"
  }
 }
}
```

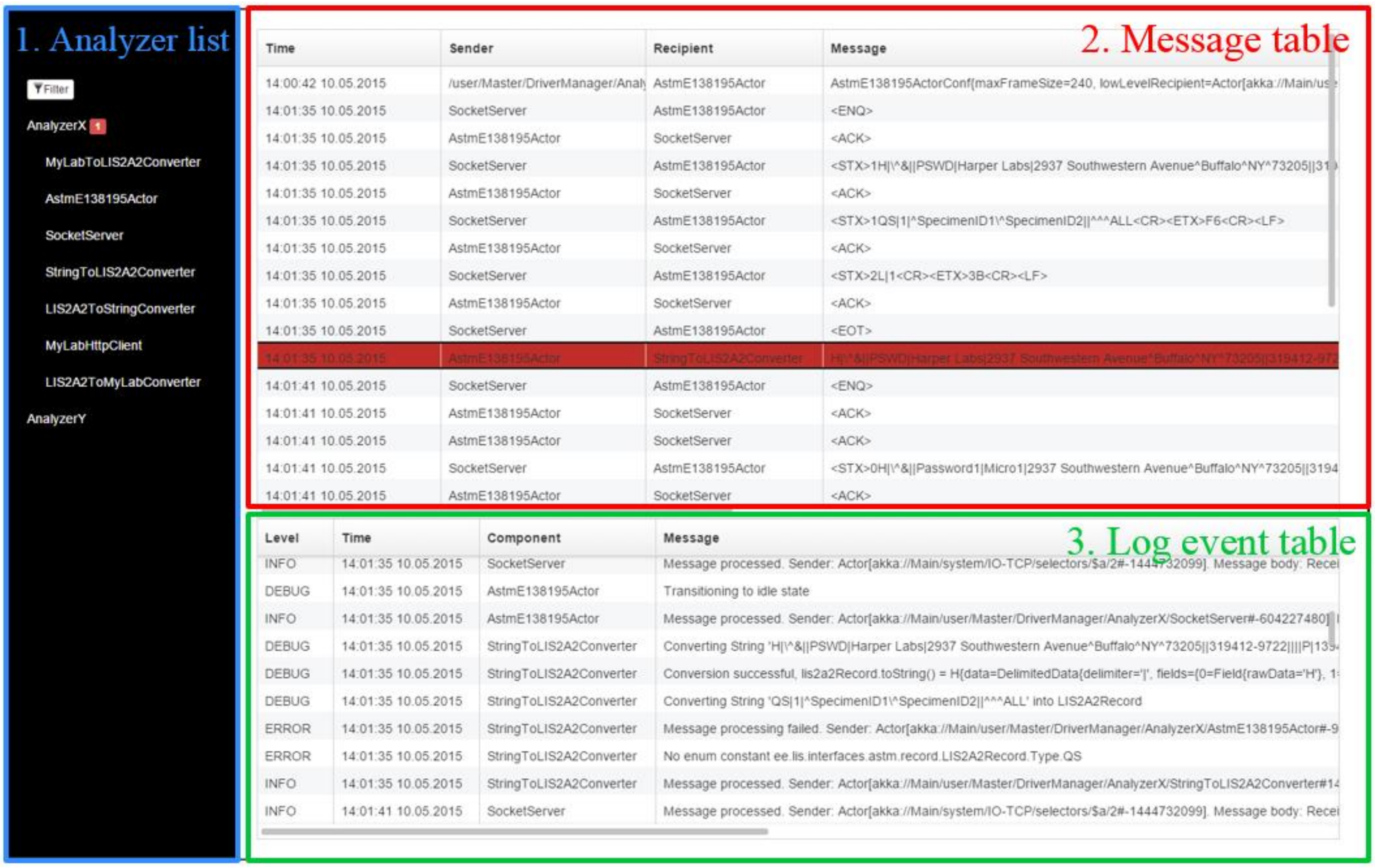
/driverDefinitions/LIS2A2OverTCP.conf

```
SocketServer {
 address = null
 port = 0
 recipientActor = "AstmE138195Controller"
}

AstmE138195Controller {
 maxFrameSize = 240
 lowLevelRecipient = "SocketServer"
 highLevelRecipient =
"StringToLIS2A2Converter"
}

StringToLIS2A2Converter {
 recipient = "LIS2A2ToMyLabConverter"
}

LIS2A2ToMyLabConverter {
 recipient = "MyLabHttpClient"
}

MyLabHttpClient {
 recipient = "MyLabToLIS2A2Converter"
 resultUrl = null
 queryUrl = null
}

MyLabToLIS2A2Converter {
 recipient = "LIS2A2ToStringConverter"
}

LIS2A2ToStringConverter {
 recipient = "AstmE138195Controller"
}
```

Listing 2: Driver configuration.

3.3. User interface

Akka does not provide a built-in feature to monitor the actors in an actor system and the messages exchanged between them. Getting an overview of the running system would mean manually shifting through log files. To provide a better overview of the system state, the user interface module was developed. It is a web application that connects to the back-end module via a websocket connection and provides an overview of the deployed drivers. Figure 6 shows a screenshot of the application.



AnalyzerX 1
- MyLabToLIS2A2Converter
- AstmE138195Actor
- SocketServer
- StringToLIS2A2Converter
- LIS2A2ToStringConverter
- MyLabHttpClient
- LIS2A2ToMyLabConverter

AnalyzerY

Time	Sender	Recipient	Message
14:00:42 10.05.2015	/user/Master/DriverManager/Anal	AstmE138195Actor	AstmE138195ActorConf(maxFrameSize=240, lowLevelRecipient=Actor[akka://Main/us
14:01:35 10.05.2015	SocketServer	AstmE138195Actor	<ENQ>
14:01:35 10.05.2015	AstmE138195Actor	SocketServer	<ACK>
14:01:35 10.05.2015	SocketServer	AstmE138195Actor	<STX>1H|\^&||PSWD|Harper Labs|2937 Southwestern Avenue^Buffalo^NY^73205||31
14:01:35 10.05.2015	AstmE138195Actor	SocketServer	<ACK>
14:01:35 10.05.2015	SocketServer	AstmE138195Actor	<STX>1QS|1|^SpecimenID1\^SpecimenID2||^^^ALL<CR><ETX>F6<CR><LF>
14:01:35 10.05.2015	AstmE138195Actor	SocketServer	<ACK>
14:01:35 10.05.2015	SocketServer	AstmE138195Actor	<STX>2L|1<CR><ETX>3B<CR><LF>
14:01:35 10.05.2015	AstmE138195Actor	SocketServer	<ACK>
14:01:35 10.05.2015	SocketServer	AstmE138195Actor	<EOT>
14:01:35 10.05.2015	AstmE138195Actor	StringToLIS2A2Converter	H|\^&||PSWD|Harper Labs|2937 Southwestern Avenue^Buffalo^NY^73205||319412-972
14:01:41 10.05.2015	SocketServer	AstmE138195Actor	<ENQ>
14:01:41 10.05.2015	AstmE138195Actor	SocketServer	<ACK>
14:01:41 10.05.2015	AstmE138195Actor	SocketServer	<ACK>
14:01:41 10.05.2015	SocketServer	AstmE138195Actor	<STX>0H|\^&||Password1|Micro1|2937 Southwestern Avenue^Buffalo^NY^73205||3194
14:01:41 10.05.2015	AstmE138195Actor	SocketServer	<ACK>

Level	Time	Component	Message
INFO	14:01:35 10.05.2015	SocketServer	Message processed. Sender: Actor[akka://Main/system/IO-TCP/selectors/$a/2#-1444732099]. Message body: Recei
DEBUG	14:01:35 10.05.2015	AstmE138195Actor	Transitioning to idle state
INFO	14:01:35 10.05.2015	AstmE138195Actor	Message processed. Sender: Actor[akka://Main/user/Master/DriverManager/AnalyzerX/SocketServer#-604227480]
DEBUG	14:01:35 10.05.2015	StringToLIS2A2Converter	Converting String 'H|\^&||PSWD|Harper Labs|2937 Southwestern Avenue^Buffalo^NY^73205||319412-9722||||P|13
DEBUG	14:01:35 10.05.2015	StringToLIS2A2Converter	Conversion successful, lis2a2Record.toString() = H{data=DelimitedData{delimiter='|', fields={0=Field{rawData='H'}, 1
DEBUG	14:01:35 10.05.2015	StringToLIS2A2Converter	Converting String 'QS|1|^SpecimenID1\^SpecimenID2||^^^ALL' into LIS2A2Record
ERROR	14:01:35 10.05.2015	StringToLIS2A2Converter	Message processing failed. Sender: Actor[akka://Main/user/Master/DriverManager/AnalyzerX/AstmE138195Actor#-9
ERROR	14:01:35 10.05.2015	StringToLIS2A2Converter	No enum constant ee.lis.interfaces.astm.record.LIS2A2Record.Type.QS
INFO	14:01:35 10.05.2015	StringToLIS2A2Converter	Message processed. Sender: Actor[akka://Main/user/Master/DriverManager/AnalyzerX/StringToLIS2A2Converter#14
INFO	14:01:41 10.05.2015	SocketServer	Message processed. Sender: Actor[akka://Main/system/IO-TCP/selectors/$a/2#-1444732099]. Message body: Recei

Figure 6: User interface.

The numbered sections serve the following purposes:

1. Analyzer list. Displays the currently connected analyzers and their driver components. In this case, *AnalyzerX* and *AnalyzerY* are connected and *AnalyzerX* is selected. If the analyzer driver has encountered errors during communication, the number of unread errors is shown next to the analyzer name. Selecting a different analyzer from the list expands the list of its components and changes the contents of tables 2 and 3 to only display data relevant to the selected driver. Clicking on the *Filter* button displays the Filter component (4).

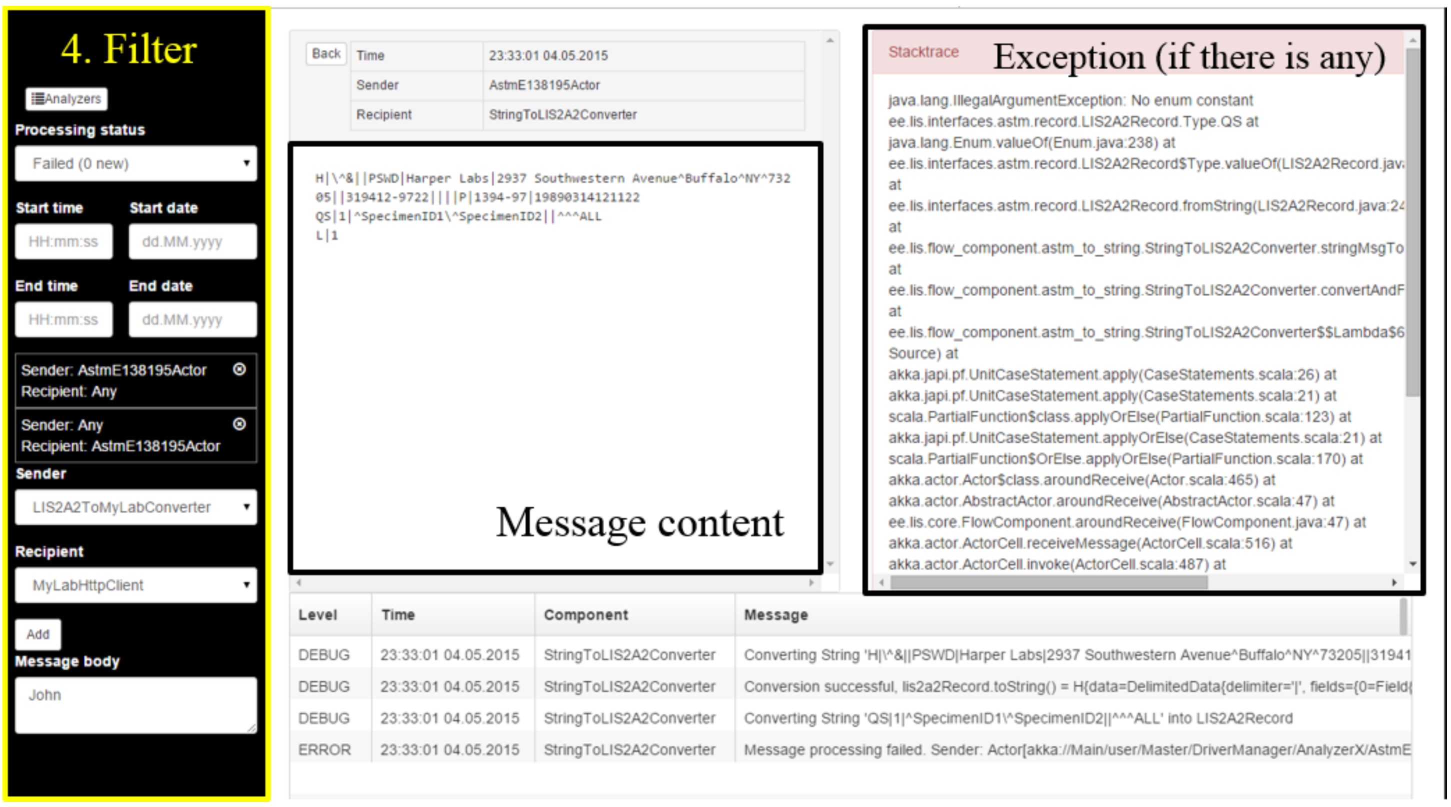
2. Message table. Displays the messages exchanged between the selected driver's components. If` processing the message failed, the row is displayed with a red background. The following information about every message is shown: date and time of message processing, sender component, recipient component, message content.

3. Log event table. Displays the log events that are triggered within the selected driver's components. These are messages that are manually added by the developer to more easily monitor and debug the components.

4. Filter (see Figure 7). The Filter component is used for searching messages in the *Message table*. It allows filtering based on the following criteria:

a) Only display messages that have been either successfully (*Successful*) processed or have caused an error (*Failed*). To ensure that all failures get checked, the number of unread failures is also displayed.
b) Only display messages that were processed within a given timeframe by specifying the Start time/date and End time/date.
c) Only display messages that were sent or received by the specified component pair(s).
d) Only display messages that contain a certain string.

Double-clicking a row in the *Message table* expands the details of this message. If the message caused an error, the stack trace of the exception is also displayed and the error is marked as read. When the detailed view of a message is opened, the *Log event table* is filtered to only display the log events that occurred during the processing of the selected message.



Processing status: Failed (0 new)

Start time	Start date
HH:mm:ss	dd.MM.yyyy

End time	End date
HH:mm:ss	dd.MM.yyyy

Sender: AstmE138195Actor
Recipient: Any

Sender: Any
Recipient: AstmE138195Actor

Sender: LIS2A2ToMyLabConverter

Recipient: MyLabHttpClient

Message body: John

Time	23:33:01 04.05.2015
Sender	AstmE138195Actor
Recipient	StringToLIS2A2Converter

```
H|\^&||PSWD|Harper Labs|2937 Southwestern Avenue^Buffalo^NY^732
05||319412-9722||||P|1394-97|19890314121122
QS|1|^SpecimenID1\^SpecimenID2||^^^ALL
L|1
```

```
java.lang.IllegalArgumentException: No enum constant
ee.lis.interfaces.astm.record.LIS2A2Record.Type.QS at
java.lang.Enum.valueOf(Enum.java:238) at
ee.lis.interfaces.astm.record.LIS2A2Record$Type.valueOf(LIS2A2Record.jav
at
ee.lis.interfaces.astm.record.LIS2A2Record.fromString(LIS2A2Record.java:24
at
ee.lis.flow_component.astm_to_string.StringToLIS2A2Converter.stringMsgTo
at
ee.lis.flow_component.astm_to_string.StringToLIS2A2Converter.convertAndF
at
ee.lis.flow_component.astm_to_string.StringToLIS2A2Converter$$Lambda$6
Source) at
akka.japi.pf.UnitCaseStatement.apply(CaseStatements.scala:26) at
akka.japi.pf.UnitCaseStatement.apply(CaseStatements.scala:21) at
scala.PartialFunction$class.applyOrElse(PartialFunction.scala:123) at
akka.japi.pf.UnitCaseStatement.applyOrElse(CaseStatements.scala:21) at
scala.PartialFunction$OrElse.applyOrElse(PartialFunction.scala:170) at
akka.actor.Actor$class.aroundReceive(Actor.scala:465) at
akka.actor.AbstractActor.aroundReceive(AbstractActor.scala:47) at
ee.lis.core.FlowComponent.aroundReceive(FlowComponent.java:47) at
akka.actor.ActorCell.receiveMessage(ActorCell.scala:516) at
akka.actor.ActorCell.invoke(ActorCell.scala:487) at
```

Level	Time	Component	Message
DEBUG	23:33:01 04.05.2015	StringToLIS2A2Converter	Converting String 'H|\^&||PSWD|Harper Labs|2937 Southwestern Avenue^Buffalo^NY^73205||31941
DEBUG	23:33:01 04.05.2015	StringToLIS2A2Converter	Conversion successful, lis2a2Record.toString() = H{data=DelimitedData{delimiter='|', fields={0=Field{
DEBUG	23:33:01 04.05.2015	StringToLIS2A2Converter	Converting String 'QS|1|^SpecimenID1\^SpecimenID2||^^^ALL' into LIS2A2Record
ERROR	23:33:01 04.05.2015	StringToLIS2A2Converter	Message processing failed. Sender: Actor[akka://Main/user/Master/DriverManager/AnalyzerX/AstmE

Figure 7: The user interface with the filter and detailed message view visible.

Chapter 4: Validation

This chapter discusses how the Akka toolkit and the implemented middleware prototype satisfy the requirements identified in Section 1.1. Each requirement is discussed in terms of how Akka can be used to address it and how it is solved in the prototype. JUnit[6] automated tests are used for requirements that can be directly measured.

4.1. Parallelization and concurrency

Parallelization

As stated in Section 2.2.3., the actor model has no shared state, which is the main reason for concurrency-related problems. Because no state is shared between driver component actors and the execution within a single actor is single-threaded, there is no need for explicit synchronization (e.g., locking). That means all drivers and their components can run in parallel.

ParallelDriversTest.java was created to verify that the proposed middleware is able to serve multiple analyzers at once. It simulates a number of analyzers being connected to the middleware and exchanging messages with the LIS simultaneously. The test was run 20 times with 10 simulated analyzers and 20 times with 1 simulated analyzer. The average times to complete the message exchanges were 12.2s and 11.5s, respectively. In all cases all messages were successfully delivered between the analyzers and the LIS. This shows that the middleware can simultaneously serve multiple analyzers without explicit synchronization or significant performance loss.

Concurrency

Race conditions caused by multiple processes accessing the same state are eliminated with the actor model. However, race conditions caused by invalid business logic can still occur. Consider the case of *AstmE1381Controller*, which sends and receives messages in a stateful manner – by exchanging a sequence of `<ACK>` messages and aggregating the received lines into a single message. The actor can be in one of the following states: sending a message, receiving a message or an idle state. Consider a scenario when the actor is in the process of receiving a message from the analyzer and at the same time also receives a message from

[6] http://junit.org/

the LIS that should be sent to the analyzer. If handled incorrectly, the middleware can start sending a message to the analyzer while the latter is also sending a message, resulting in the failure of both parties to deliver their messages.

Akka provides a built-in feature called *stashing* for handling these kinds of use cases. An Akka actor provides the methods *stash()* and *unstashAll().* The *stash()* method places the message in a waiting list and the *unstashAll()* method moves the stashed messages back to the actor's mailbox. For example, when the *AstmE1381Controller* actor is in the process of receiving a message from the analyzer, it can choose to *stash* all messages not sent from the analyzer. Once the exchange sequence with the analyzer has been completed and the actor transitions to an idle state, it can *unstash* the messages in the waiting list and continue processing them.

AstmE138195ControllerTest.java was implemented to verify the validity of this approach. It simulates the same scenario, where the *AstmE1381Controller* is receiving messages from both the LIS and analyzer at the same time. Running the test shows that all messages get delivered to both the LIS and the analyzer.

4.2. Fault tolerance

Errors in a driver's execution mainly occur when the programmer's assumptions of a message format or delivery protocol do not hold (e.g., incorrectly assuming the presence of a data field and generating a *NullPointerException* by trying to read it).

Two requirements were considered when designing the prototype for fault-tolerance:

- An error within the execution of one driver should not affect other running drivers.
- A driver should recover from an error – the invalid message should be logged and the driver should continue processing the next message.

Akka ensures that an error occurring in one actor is isolated from the rest of the actor system. The supervision architecture provides an effective way for handling errors and not leaving the application in an invalid state. When an error occurs in one of the driver's component actors, it is restarted by its parent (*DynamicDriver*) and continues processing the next message while not affecting other actors in the system.

FaultToleranceTest.java was implemented to verify that the two requirements are met. Similarly to the parallelization test, it simulates 5 analyzers that are simultaneously exchanging messages via the middleware:

- *faultyAnalyzer* – simulates both valid and invalid messages sent from the analyzer.

- *analyzerWithFaultyLIS* – simulates an analyzer that sends valid messages, but receives invalid responses from the LIS.
- *mockAnalyzer1-3* – remaining 3 analyzers that send valid messages and receive valid messages from the LIS.

The expected behaviour is that all valid messages are delivered, the invalid messages are skipped and the errors are logged. Running the test verifies that the system behaves as expected – errors caused by the invalid messages have no effect on delivering the valid messages.

4.3. Monitoring

Since Akka does not provide a built-in feature to monitor the actors and the messages exchanged among them, the user interface module was developed. The following requirements were considered during implementation:

Detection of errors. When an error occurs, it should be easily detectable from the user interface.

The user interface highlights any messages that have caused an error so that they are easily spotted in the *Message table* (see Section 3.3.). In addition, it keeps track of the unread error count. When an error occurs, it is marked as “unread” and the unread error count is displayed next to the analyzer name in the *Analyzer list*. This ensures that the user has an overview of whether any new errors have occurred.

Identifying the cause of the error. The cause of errors should be traceable in retrospect, because reproducing an error can be complicated as it usually requires testing on real analyzers.

This is achieved by displaying the stack trace of the thrown exception in the *Message table*. This allows the developer to identify the line of code that threw the exception. In combination with the message content that caused the error and the log events that occurred while processing the message, the cause can be identified.

Searching the message history. If the system is not behaving as expected, the message history has to be observed find the cause. To ease the troubleshooting process, it is beneficial to provide functionality for searching messages. For example, if the analysis results for a specific patient have not reached the LIS, then searching for messages containing the patient name can help locate the problem.

This functionality is provided by the *Filter* in the sidebar. It enables to filter the content of the *Message table* based on message processing status, time of message, the sender and receiver components and message content.

4.4. Code reusability

The middleware implementation adopts a component-based model with components being the actors that can be linked to form a driver for a specific analyzer. The following characteristics of reusable components were considered when designing for reusability [14]:
High cohesion. Cohesion shows how well-defined is the purpose of a component. It is the notion of how related is the functionality offered by the component. A component that has a more focused purpose is more cohesive than a component that tries to do many unrelated things.

As discussed in Section 3.2.1., a driver is made up of component actors, each performing a single step of the message processing. The reason for splitting the message processing into a pipeline is to achieve high cohesion. As each component actor in the pipeline is only responsible for a single well-defined task, the cohesion of the components is increased.
Low coupling. Coupling shows how dependent a component is on others. A component that requires five other components to function is more highly coupled than a completely self-contained component. Highly coupled components are difficult to modify, because changes in one component often require changes in other components.

A single driver component has no knowledge of the other components within a driver and thus is not bound to a particular driver. It only knows the address of the component that it should forward the processed message to, which is specified via the runtime configuration (see Section 3.2.2.). This lowers the level of coupling as different drivers can be composed from existing components by changing the configuration file and without making changes to the components themselves. A generic component can be used in different drivers just by instantiating it with a different recipient component address.

However, since the components need to communicate, they cannot be completely decoupled. The interface for a component is the types of messages it can process. Making changes to this interface may also require changes to other components. For example, if a message type is removed from a component, all components that depend on the component being able to process that message type, have to be modified accordingly. As long as this interface is unchanged, the internal implementation of components can be changed without affecting others.

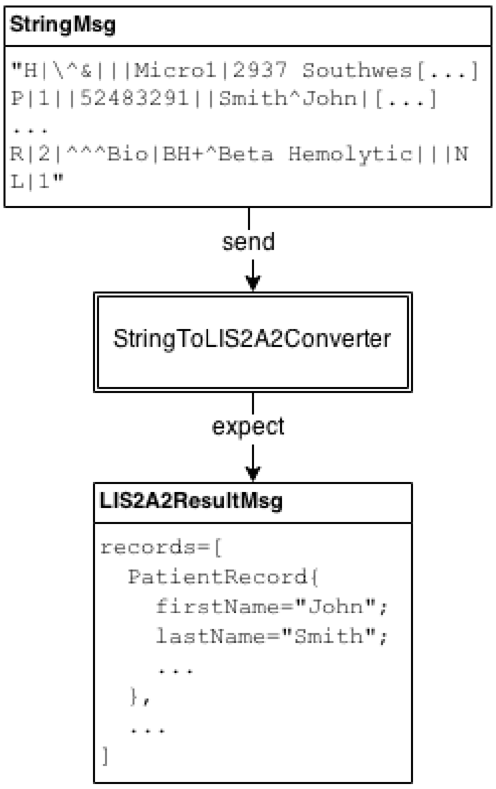
Consider the sample driver implemented in this thesis (Figure 5). It communicates with the analyzer using the LIS2-A2 data format. The data transfer protocol is ASTM E1381-95 over a TCP connection. Consider a case where a new analyzer needs to be connected that communicates the same data format over the local file system (i.e., writes and reads text files from a directory). This can be achieved by implementing a component that reads and writes files to a directory. As long as it sends and is able to process messages of type *String*, the *AstmE138195Controller* and *SocketServer* components can be replaced by it using the runtime configuration file without modifying the source code of any of the other components. Similarly, the current driver implementation can be modified for another LIS by implementing replacements for only the MyLab-specific components.

4.5. Testability

When writing automated tests for the middleware, the developer should be able to test different levels of functionality.

4.5.1. Testing individual components

These tests are used to verify the behaviour of an individual component actor. For example, the *StringToLIS2A2Converter* component can be tested by sending it a *String* and verifying that the *LIS2A2Msg* it sends out was correctly constructed (Figure 8).



```
"H|\^&|||Micro1|2937 Southwes[...]
P|1||52483291||Smith^John|[...]
...
R|2|^^^Bio|BH+^Beta Hemolytic|||N
L|1"
```

```
records=[
  PatientRecord{
    firstName="John";
    lastName="Smith";
    ...
  },
  ...
]
```

Figure 8: Testing an individual actor.

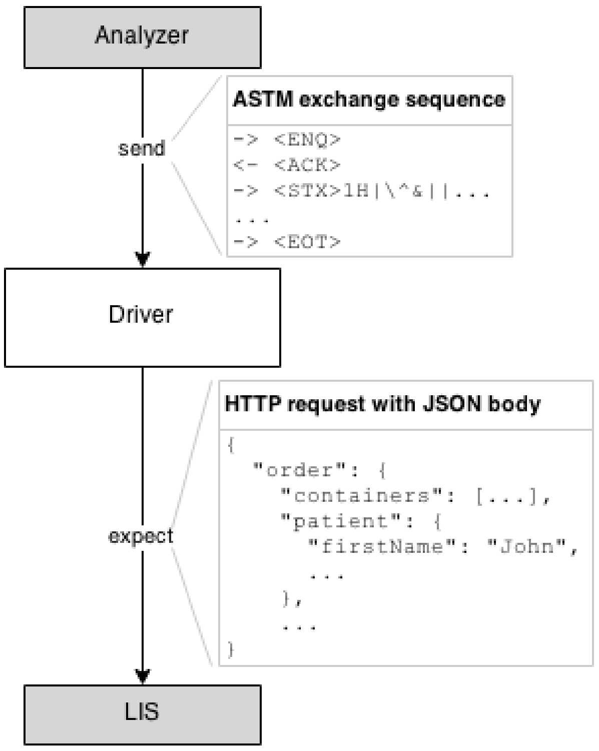
The *akka-testkit* library provides features for testing actor systems. One of them is the *JavaTestKit* class, which is an actor that allows to examine the messages it receives. It acts as a probe that receives the output of the component under test and verifies its content. Listing 3 shows a part of *StringToLIS2A2ConverterTest.java* that uses *JavaTestKit* to verify the behaviour of the *StringToLIS2A2Converter* component.

```
@Test
public void resultStringToLIS2A2ResultMessageTest() {
  //create the actor under test (AUT)
  final ActorRef actorUnderTest = system.actorOf(Props.create(StringToLIS2A2Converter.class));
  //create a probe and tell the AUT to forward its messages to the probe
  final JavaTestKit probe = new JavaTestKit(system);
  stringToLIS2A2Converter.tell(new RecipientConf(probe.getRef()), ActorRef.noSender());
  //specify the input and expected output for the AUT
  String inputString =
    "H|\\^&||Password1|Micro1|2937 Southwestern [...] |19890501074500" + CR +
    "P|1||52483291||Smith^John|Samuels|19600401|M|W|4526 C [...]" + CR +
    "O|1|5762^01||^^^BC^BloodCulture^POSCOMBO| [...] |Instrument#1" + CR +
    "R|1|^^^Org#|51^Strep Species|||N" + CR +
    "R|2|^^^Bio|BH+^Beta Hemolytic|||N" + CR +
    "L|1" + CR;
  LIS2A2ResultMsg expectedOutput = [...]; // manually construct a LIS2A2ResultMsg
  //send the input to the AUT
  stringToAstmConverter.tell(inputString, ActorRef.noSender());
  //order the probe to expect a message (the output of the AUT)
  LIS2A2ResultMsg received = probe.expectMsgClass(LIS2A2ResultMsg.class);
  //after the message has been received, verify that the content matches the expected output
  Assert.assertEquals(expected, received);
  Assert.assertEquals(inputString, received.asString());
}
```

Listing 3: Testing a single actor with *JavaTestKit*.
Parts of text are ommited with „[...]" for formatting purposes.

4.5.2. Testing drivers

Another level of functionality to test is the whole driver, i.e., the combination of cooperating component actors (Figure 9). This ensures that when changing an individual component, all dependent drivers retain their functionality.



```
-> <ENQ>
<- <ACK>
-> <STX>1H|\^&||...
...
-> <EOT>
```

```
{
  "order": {
    "containers": [...],
    "patient": {
      "firstName": "John",
      ...
    },
    ...
}
```

Figure 9: Testing a driver.

Listing 4 shows a part of *LIS2A2OverTCPTest.java*, which was implemented to test the example driver in this thesis. It uses two helper actors:

- *mockAnalyzer* – a TCP client that connects to the driver and mimics the analyzer using the ASTM low level protocol.
- *mockMyLab* – an HTTP server that can receive HTTP requests and mimics the MyLab LIS.

Running the test starts the middleware with the */driverDefinitions/LIS2A2OverTCP.conf* configuration file. Next, *mockAnalyzer* starts an ASTM low level exchange sequence with the driver and *mockMyLab* waits for an HTTP request containing the information sent by *mockAnalyzer* in the MyLab-specific format. This test is treats the driver as a black box and is not concerned with individual steps in the message processing pipeline. It verifies that the endpoints of the driver (the analyzer and LIS) receive the correct messages.

Besides verifying the functionality of the driver, the test also serves as documentation for the implemented driver. It gives the reader a high-level overview of the driver protocol ithout having to examine how the driver is implemented in terms of individual components.

```
@Test
public void receiveResult() {
  mockAnalyzer
    .send("<ENQ>")
    .expect("<ACK>")
    .send("<STX>H|\\^&||Password1|Micro1|2937 Southwest [...] <CR><ETX>{CS}<CR><LF>")
    .expect("<ACK>")
    .send("<STX>P|1||52483291||Smith^John|Samuels|19600401 [...] <CR><ETX>{CS}<CR><LF>")
    .expect("<ACK>")
    .send("<STX>O|1|5762^01||^^^BC^BloodCulture^POSCOM [...] <CR><ETX>{CS}<ETX><CR><LF>")
    .expect("<ACK>")
    .send("<STX>R|1|^^^Org#|51|||N<CR><ETX>{CS}<CR><LF>")
    .expect("<ACK>")
    .send("<STX>R|2|^^^Bio|BH+|||N<CR><ETX>{CS}<CR><LF>")
    .expect("<ACK>")
    .send("<STX>L|1<CR><ETX>4A<CR><LF>")
    .expect("<ACK>")
    .send("<EOT>")
    .expectNoMsg();
  mockMyLab
    .expect(
      MyLabResultMsg(
        Order(
          Patient("John", "Smith", "52483291"),
          containers(
            Container("5762",
                      analyses(
                        Analysis("Org#", "", Result("51", "")),
                        Analysis("Bio", "", Result("BH+", ""))))))))
    .send("");
}
```

Listing 4: Example of a driver test (*LIS2A2OverTCPTest*.java).
Parts of text are ommited with „[...]“ for formatting purposes.

Chapter 5: Conclusion

This thesis addressed the problem of connecting medical analyzers with laboratory information systems. It introduced the concept of laboratory middleware and its requirements. The actor model and its Akka implementation were also introduced. The aim of the thesis was to find out whether Akka is suitable for implementing laboratory middleware.

An Akka-based middleware prototype was developed with the goal of satisfying these requirements. The prototype also includes a reference implementation that shows an approach of using Akka for developing interfaces between an analyzer and a LIS. This approach can be used as a basis for interfacing different analyzers and LISs.

The thesis is concluded with a discussion about how Akka and the implemented prototype satisfy the identified requirements of laboratory middleware.

5.1. Limitations

The implemented middleware prototype has limitations that should be addressed before considering using it in a production environment. Firstly, there are security concerns – the back-end module shares the message history with any client that connects to it. This poses a security risk as the messages contain patient credentials and medical information. Secondly, the message history is currently memory-based. This means there is a limit on the number of messages that can be viewed from the user interface. The history is also lost when the middleware is restarted. This can become a problem when it is necessary to view messages from a past period.

5.2. Future work

As the implemented middleware is only a prototype, there are many possible improvements.

The current solution has no security features. Therefore, additional development is required to make a middleware instance accessible only to the authorized personnel.

The current monitoring feature is memory-based. This makes it not suitable for high-throughput scenarios, as the memory for storing message history is limited. A persistent storage mechanism (e.g., a database) for the message history would enable to store message history for a longer time period. That would also allow to buffer messages, if they cannot be forwarded immediately (e.g., if the LIS is unavailable).

The configuration management is currently based on local text files. A form-based configuration management solution would be more user-friendly than editing the syntax-sensitive configuration files. Also, a remote way to manage the configuration would help troubleshoot problems more quickly, as the developers are not usually present at the laboratory and would otherwise have to rely on the laboratory staff sending configuration files back and forth.

Bibliography

1. Terry, M. (2011): Transferring Laboratory Data Into The Electronic Medical Record: Technological Options For Data Migration In The Laboratory Information System.
2. Selmeyer, J., Cloutier, B. (1996): Interfacing the Clinical Laboratory: A Primer for LIS Managers
3. Hewitt, C., Bishop, P., Steiger, R. (1973): A Universal Modular ACTOR Formalism for Artificial Intelligence.
4. Hewitt, C. (2010): Actor Model of Computation: Scalable Robust Information Systems.
5. Armstrong, J. (2003): Making reliable distributed systems in the presence of software errors.
6. Typesafe Inc (2015): Akka Java Documentation. http://doc.akka.io/docs/akka/2.3.9/AkkaJava.pdf
7. Erb, B. (2012): Concurrent Programming for Scalable Web Architectures.
8. Mirth Connect - http://www.mirthcorp.com/products/mirth-connect
9. Zhu, J. (2005): Automatic laboratory operations by integrating laboratory information management systems (LIMS) with analytical instruments and scientific data management system (SDMS).
10. Froid. Free Open Instrument Middleware. https://github.com/bikalabs/Bika-LIMS/wiki/GSoC-%C2%B7-Froid.-Free-Open-Instrument-Middleware
11. ASTM E1381-95 Standard http://www.astm.org/DATABASE.CART/HISTORICAL/E1381-95.htm
12. Clinical and Laboratory Standards Institute: Specification for Transferring Information Between Clinical Laboratory Instruments and Information Systems: Approved Standard - Second Edition.
13. Roestenburg, C., Bakker, R., Williams, R. (2015): Akka in Action (MEAP edition, version 15).
14. Sametinger, J. (1997): Software Engineering with Reusable Components.

Appendix A: Running the middleware and automated tests

Source code

The source code is included as a zip file in the extras that are submitted with this thesis. It can also be accessed from the following repository: https://github.com/teoreteetik/aesop

Running the middleware

Back-end module

The packaged back-end module is available in */backend/dist/dist.zip*. The zip archive contains two files: *middleware-1.0.jar* and *application.conf*. File *application.conf* contains the configuration for the middleware. By default, it contains the configuration to start the example driver on port 50000 and the websocket server for the user interface on port 8900. The middleware can be run from the command line using the following command (Java 8 required):

```
java -jar middleware-1.0.jar
```

User interface

The packaged user interface is located in */ui/build/index.html*. When this file is opened, the application first asks for the websocket address to connect to the back-end module. The default address is *ws://127.0.0.1:8900*. After pressing the *Connect* button, the connection to the back-end module is established and the user interface is displayed.

Usage example

To simulate an analyzer and LIS communicating via the middleware, follow these steps:

1. Start the back-end module and user interface (see above).
2. Run the following command in the */backend* directory:

   ```
   gradlew test --tests *LIS2A2OverTCPDemo
   ```

   After running this command, the message exchange should be displayed in the user interface.

Ports 50000 and 8070 should be free as these are used by the mimicking analyzer and LIS.

Running automated tests

The tests used in Chapter 4 can be run from the command line using Gradle[7]. When running a test for the first time, a local copy of Gradle is downloaded automatically. The commands for running the tests are presented in the table below. These commands should be run in the */backend* directory. Ports 50000-50010 and 8070, 8071, 8900 should be free for running the tests, because these ports are used for mimicking the analyzers and LIS in some of the tests. The test reports are saved to */backend/build/reports/tests*.

ParallelDriversTest	`gradlew test --tests *ParallelDriversTest.parallelDriversTest`
AstmE138195ControllerTest	`gradlew test --tests *AstmE138195ControllerTest.parallelTest`
FaultToleranceTest	`gradlew test --tests *FaultToleranceTest.faultToleranceTest`
StringToLIS2A2ConverterTest	`gradlew test --tests *StringToLIS2A2ConverterTest`
LIS2A2OverTCPTest	`gradlew test --tests *LIS2A2OverTCPTest`

[7] http://gradle.org/