FH JOANNEUM - University of Applied Sciences

Actor Based Business Process Execution via Intelligent Business Objects

Master thesis

submitted at the Master Degree Programme Information Management

for the degree „Diplomingenieur für technisch-wissenschaftliche Berufe“

„Diplomingenieurin für technisch-wissenschaftliche Berufe“

Author:

Florian Unterberger

Supervisor:

Robert Singer

Graz, September 2016

FH | JOANNEUM
University of Applied Sciences

Obligatory signed declaration:

I hereby declare that the present master's thesis was composed by myself and that the work contained herein is my own. I also confirm that I have only used the specified resources. All formulations and concepts taken verbatim or in substance from printed or unprinted material or from the Internet have been cited according to the rules of good scientific practice and indicated by footnotes or other exact references to the original source.

The present thesis has not been submitted to another university for the award of an academic degree in this form. This thesis has been submitted in printed and electronic form. I hereby confirm that the content of the digital version is the same as in the printed version.

I understand that the provision of incorrect information may have legal consequences.

_____

Florian Unterberger Graz, 19.09.2016

Table of Contents

Table of Figures

Table of Listings

Table of Tables

Table of Abbreviations

IT	Information Technology
BPMN	Business Process Model and Notation
BPEL	Business Process Execution Language
S-BPM	Subject-oriented Business Process Modelling
API	Application Programming Interface
IBO	Intelligent Business Objects
OOP	Object-Oriented Programming
ID	Identification Number
OTP	Open Telecom Platform
GUI	Graphical User Interface

Abstract

Business processes are a vital part of every company. But executing business processes with conventional business process engines is complex, and so people rely on services such as email, where processes naturally emerge from the users' interactions.

Instead of relying on the typical centralised and monolithic architecture of business process engines, this thesis tries to show, that a decentralised approach utilising the Actor Model and Intelligent Business Objects (IBOs) is also feasible. Using the Actor Model, the process engine is distributed on several smaller independent actors, which enables a high flexibility. But instead of needing to store process configurations on actors to execute a process, the process configuration is stored in an Intelligent Business Object, which is passed around by the actors for each process instance. As each actor can execute its part of the process without any pre-configuration, a process can even be updated while it is already executing.

Starting with a basic architecture, the system of actors is gradually expanded, until business processes that allow human interactions can be executed. The architecture is then implemented using the programming language Erlang to create a working prototype. The approach is considered feasible when processes based on the Actor Model and IBOs can be set up, executed and interacted with using a graphical user interface.

As the prototype fulfilled the requirements as mentioned above, the approach is viable. Furthermore, the graphical representations of IBOs and the Business Process Model and Notation (BPMN) are both graph oriented and similar, which might allow conversions between the two.

How well this new approach fares against existing solutions has yet to be determined, as its current limited implementation does not permit a direct comparison yet. But this work could open up a new chapter in how business processes are executed.

Kurzfassung

Geschäftsprozesse sind unverzichtbar für jede Firma. Aber Geschäftsprozesse tatsächlich auszuführen ist aufwändig, weshalb viele Leute oft auf Dienste wie Email zugreifen, wo Prozesse einfach durch Nutzerinteraktion entstehen.

Anstatt der üblichen zentralen und monolithischen Architektur von Geschäftsprozessengines versucht diese Arbeit zu zeigen, das eine dezentrale Herangehensweise auch möglich ist, durch die Verwendung des Aktorenmodells und Intelligenter Geschäftsobjekte (IBOs). Das Aktorenmodell erlaubt es, die Geschäftsprozessengine auf mehrere Aktoren aufzuteilen, wodurch eine höhere Flexibilität ermöglich wird. Aber anstatt die Prozesskonfiguration auf den Aktoren zu speichern, wird die Prozesskonfiguration in einem Intelligenten Geschäftsobjekt gespeichert, welches von den Aktoren herumgereicht wird für jede Prozessinstanz. Da jeder Aktor seinen Teil des Prozesses ohne Vorkonfiguration ausführen kann, kann ein Prozess auch während seiner Ausführung verändert werden.

Beginnend mit einer Basisarchitektur wird das System der Aktoren schrittweise erweitert, bis Geschäftsprozesse ausgeführt werden können, die eine Interaktion mit Menschen erlauben. Anschließend wird die Architektur implementiert unter Verwendung der Programmiersprache Erlang, um einen funktionierenden Prototypen zu erstellen. Das Konzept wird als brauchbar erachtet, wenn Prozesse basierend auf dem Aktorenmodell und IBOs erstellt, ausgeführt und interagiert werden können mithilfe eines grafischen Benutzeroberfläche.

Da der Prototyp die oben genannten Anforderungen erfüllen konnte, ist das Konzept umsetzbar. Zusätzlich sind die grafische Darstellung der IBOs und der Business Process Model and Notation (BPMN) beide Graph-orientiert und ähnlich, was eine mögliche Konvertierung zwischen den beiden erlauben könnte.

Wie gut dieser neue Ansatz gegen bestehende Lösungen abschneidet muss noch festgestellt werden, da die derzeitige limitierte Implementierung noch keinen direkten Vergleich erlaubt. Aber diese Arbeit könnte möglicherweise ein neues Kapitel öffnen, wie Geschäftsprozesse ausgeführt werden.

1 Introduction

Business processes are the foundation of every successful company. Well designed and executed processes enable companies to be deliver the best value to their customers and other stakeholders. Therefore, businesses focus on constantly improving their processes. Because of this importance business process management has a high priority in business administration communities. [1] [2]

Another big driver of business processes are information technologies, which further enhance the performance of business processes. The importance of IT is self-evident, as electronic data processing happens in the blink of an eye that cannot be matched with human resources. But IT does not only automate tasks, that had to be done manually before, but also extends human capabilities, such as enabling us to communicate instantly without the need to be in the same room. It is therefore obvious that IT is a big factor in business processes.

Both business administration and computer science communities have the same task at hand, but they look at processes from different points of view because of their different background. The computer science community is further divided into researchers, trying to describe processes formally and software developers, trying to develop software that can be used. Because of these different backgrounds, a gap exists between IT and business people, which creates issues when trying to execute business processes. [2]

1.1 Process models and execution

Business analysts typically approach processes from a conceptual point of view by creating models of processes. But a process model alone cannot be executed on a computer. So the task to cast a process model into something that can be executed on a computer is then left to software developers, who translate the concept into a working software, which is in turn executed (Figure 1). This is not very efficient, which is why a lot of emphasize has been put into improving the pipeline between the process model and the execution.

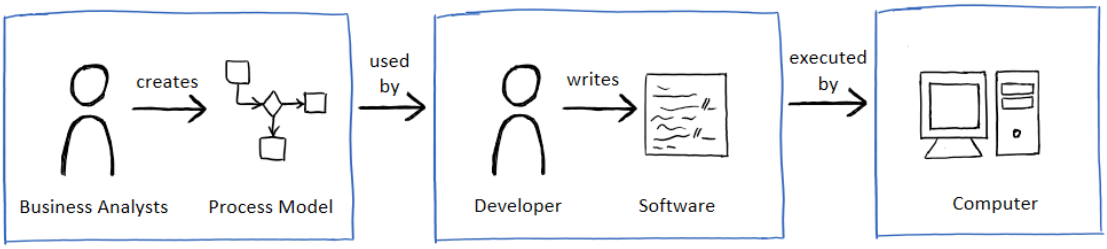

Figure 1 - From business model to executable software

The ultimate goal of business process management is to execute the process model straight away, but that is easier said than done. The de-facto standard for business process modelling is the Business Process Model and Notation (BPMN) [3], but the standard alone is not executable. In other words, the standard is too vague to be executed directly. Some software vendors have therefore extended the standard and made BPMN models executable, but only a subset of the BPMN standard's symbols

are supported. As a result, users have to change the way they used to model BPMN models, as they only have certain symbols available for modelling [4].

For process execution, the Business Process Execution Language (BPEL) was created. Unlike BPMN process models, BPEL process models can be executed, but BPEL is not used by business analysts to model processes. Therefore, developers translate process models that they receive in the BPMN standard to a model in BPEL [5] (Figure 2), but this is not very efficient.

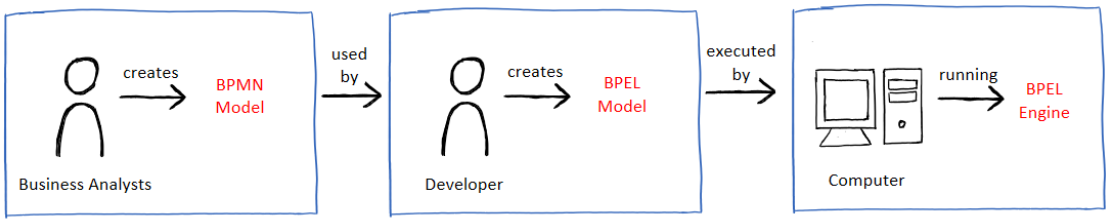

Figure 2 - From BPMN model to executable via BPEL model

A solution to the efficiency problem from Figure 2 would be if the transformation between BPMN and BPEL could be done automatically, but this is not just a simple exercise, as there are conceptual differences between these two standards. Not all patterns of the graph-oriented BPMN can be matched directly with the block-oriented BPEL, which still creates limitations when transforming BPMN models into BPEL ones. [6]

If business analysts would use BPEL instead of BPMN, then no intermediary step would be needed. But as time showed, business analysts use BPMN. Therefore, BPMN must have advantages when modelling processes compared to BPEL. This leaves room for further improvements. A standard which tries to fill that room is S-BPM (Subject-oriented Business Process Modelling), which simplifies modelling, while it is also directly executable [7].

1.2 Emerging processes and workflows

If processes are unspecified, then processes emerge automatically from human interaction. It is not necessary to specify processes, albeit an emerged process can lack transparency and might be inefficient and even violate other company policies. But nonetheless, not every single process in a company can be defined, as either a process is too information centric (a defined process would impede knowledge workers) or only occurs very seldom (defining a process would take more time than a slightly inefficient emerging process).

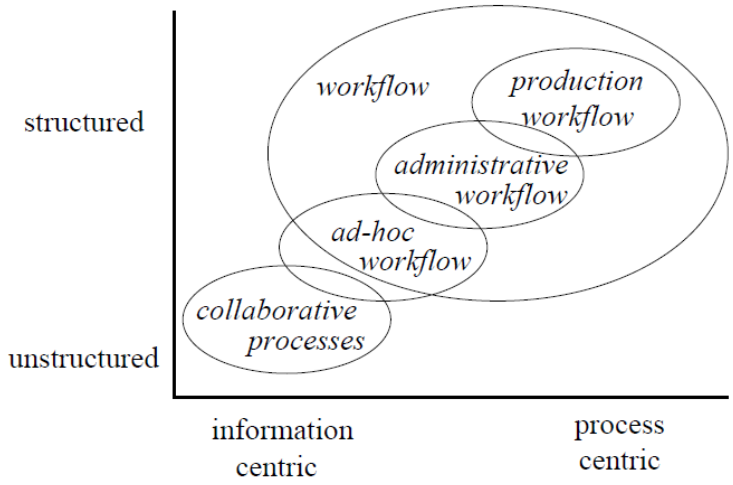

Figure 3 - Workflow processes versus collaborative processes, from: [8]

Figure 3 categorizes processes into workflows and collaborative processes. The difference is essential: workflows are very explicitly defined, while collaborative processes are only loosely specified [8]. The more a business process is defined, the easier it is to execute them, which is why specialised workflow software is utilised for manufacturing. Processes, which are not well defined, are much more difficult to execute, as computers cannot handle the uncertainty on their own and require clear instructions.

As not all processes can be defined in detail and in advance, employees rely on direct communication, which can be seen as ad hoc processes. This can be done for instance face to face, via calls or email. The email service is particularly interesting, as it only contains basic functionality (send to one/many, reply to one/many) but can be leveraged and used like a business process. A person writing a single email does not yet make a process, but the interaction that follows can already be seen as a highly dynamic process. Additionally, the email system only consists of independent mail boxes and distributed mail servers, with no central point of execution, unlike traditional process engines [9].

As email is already used on a day to day basis for ad hoc processes, why not use its principles as the basis for more formalized business process execution? This thesis tries to show that this might be a good idea and utilises the Actor Model for this purpose, as the email system can be expressed with the Actor Model.

1.3 Actor Model

Carl Hewitt, one of the three creators of the Actor Model in 1973 [10], defines the model as *"[…] a mathematical theory of computation that treats "Actors" as the universal primitives of digital computation."* [11]. But this definition is incomplete, as it does not explain what actors are.

A more complete definition was given by Vaughn Vernon:

> *The Actor model promotes actors as the first-class unit of computation and emphasizes communication between actors via message sending. Because message sending is asynchronous, actors operate in a highly concurrent manner, which naturally makes for problem solving in parallel. Each actor generally cares for a single application responsibility. [12]*

In other words, actors are completely independent parts of an application. On its own, a single actor only has a limited functionality. But by communicating via messages between each other, they form complete enterprise applications. Thus, whenever an actor receives a message, the actor can react in three distinctive ways:

- Send one or more messages to other actors
- Create one or more new actors
- Change its behaviour for the next received message

The emphasize is on "can", because the actor decides on its own what to do with the message. An actor can, for example, also choose to not react to a message at all, to only react with an internal state change or to react in all three ways at the same time. [11], [12]

1.3.1 Characteristics and Advantages

The actor model only became more known recently, even though the actor model is already over 40 years old. The reason for that is that only now we are capable of utilising the big advantages of the model. At the time the model was introduced, computer hardware was simply not advanced enough to profit from the model's characteristics. [12]

The following basic characteristics with their advantages are a summary of [12]:

- *Asynchronous messaging*: Actors communicate by passing messages asynchronously. That way actors can run on separate threads and do not need to wait for a response (= the thread does not lock).
- *State Machines*: By changing their internal characteristics, actors can also be used as a finite state machine.
- *Share nothing*: Every actor has its own mutable state.
- *Lock-free concurrency*: Actors do not need any locking mechanism, as they do not share their mutable state and messages are received one by one.
- *Parallelism*: Parallelism is similar to concurrency. Concurrency can be seen as independent tasks running next to each other, while parallelism is referring to multiple actors working together simultaneously to complete a single task.
- *Actors come in systems:* As previously described, single actors only have a limited functionality, but the sum of all actors create whole applications. This is also expressed in the philosophy of actor systems, where essentially any problem can be solved by just adding another actor.

All these characteristics are virtually predestined for today's multi-core processors, showing that the actor model was truly ahead of its time.

1.3.2 Email system as Actor Model

The following example of the email system in Figure 4 shows, how the Actor Model can be used to describe the email communication, including the implications of using asynchronous messages as a means of communication. Mail accounts with its mailboxes serve as actors, while email addresses are the corresponding actor addresses. Emails are the messages that are send between the actors. [11]

1. Maria wants to send an invitation to Werner and Clara, but she does not know their addresses. But she knows, that Peter has their addresses, so she asks him to forward her invitation to Werner and Clara. Maria also does not know, when Peter will actually read the email. Peter could also just ignore her email as well. She would only know, if Peter would reply to her message.
2. Peter then reads the email and forwards it to Werner and Clara. Like Maria before, Peter does not know when Werner and Clara will read the email and how they will react to it. Instead of writing an email to Clara and Werner, Peter could have also informed Maria about their addresses, so that Maria could contact them directly.
3. As soon as Werner and Clara read the email, the initial purpose of sending an invitation to Werner and Clara is fulfilled. If Werner or Clara would contact Maria, Maria could deduce that Peter forwarded the email as she asked him to.

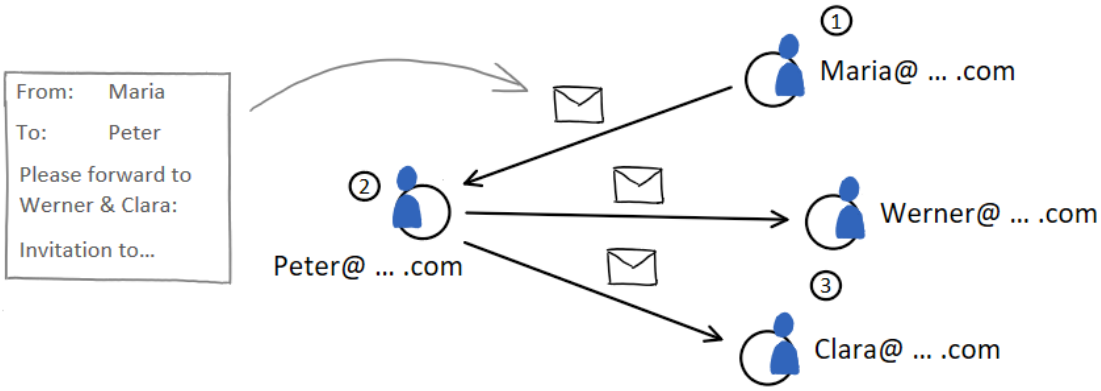

Figure 4 - Email system as Actor Model

The implications of the example above also apply to the actor model in general. Like Maria, actors also need to know the destination address of the messages. In case the destination address is not known to an actor, other actors can be used to forward messages or as an alternative the destination address can be requested from another actor. Similar to normal human people, the internal behaviour of actors cannot be observed by other actors. Actors can only infer the behaviour of other actors by their messages. Nevertheless, knowing how other actors work in detail is not necessary for actors, as they only need to know about the others' external behaviours[1], which hides the complexity[2] of the overall system.

[1] From a programmer's point of view, the external behaviour can be referred to as application programming interface (API).
[2] Similar to the encapsulation of objects with accessing methods in object-oriented programming.

1.4 Leveraging the actor model for business process execution

The email example from the previous chapter already indicated the potential of using messages and actors to execute business processes. Not only do actors work independently of each other, they can also be deployed very flexibly. Hiding their internal behaviour from each other enables them to rely and integrate even very complex tasks of other actors. But the important question is, how actors can be configured to work with each other, because it is necessary to readjust how the actors interact with each other for each single business process. The following chapters showcase three possible ways, how the actors' behaviour can be adapted to execute processes.

1.4.1 Central business process engine

Arguably the simplest way would be to use a central business process engine, which relies on other actors to offload certain functionalities (Figure 5). But this approach is not really an improvement over the prevalent architecture, as extensions in regular business process engines essentially fulfil the same role.

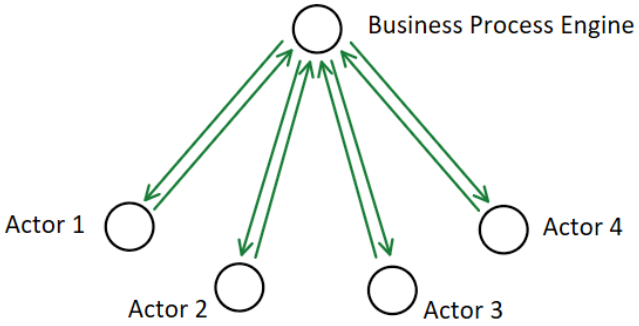

Figure 5 - Actor architecture with a central business process engine

1.4.2 Preconfigured actors

Another possible way would be to remove the central business process engine altogether and instead split the activities between the actors for each process. This way, the work of a process instance is split between several actors, instead of a single central engine. But before the process can be executed, the involved actors have to be configured beforehand (Figure 6). Thus, process configurations have to be stored in each actor's internal configuration.

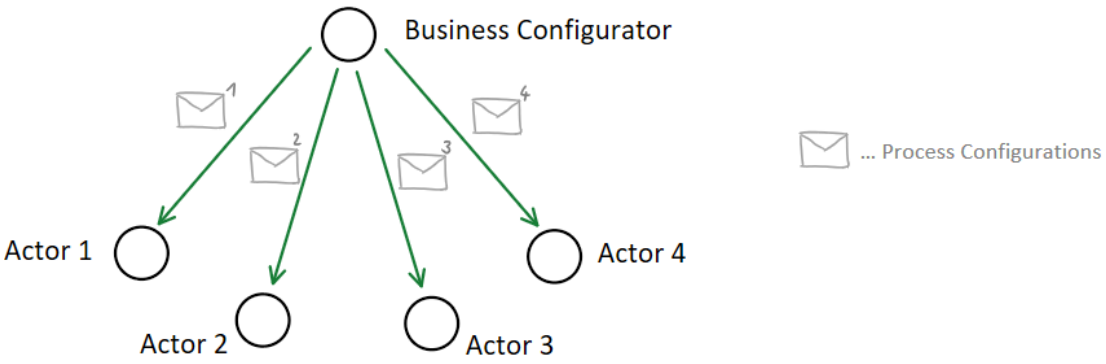

Figure 6 - Preconfiguration of actors

Then, when the preconfigured actors receive a certain message, they know what to do (Figure 7) and can communicate between themselves, instead of relying on a central engine. The messages, which are exchanged between the actors, are so called business objects, which are pieces of data, linked to specific business process instances[3].

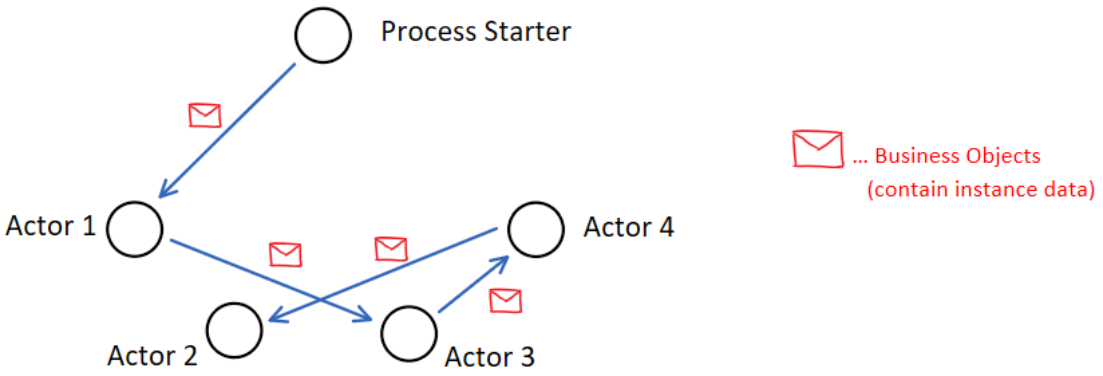

Figure 7 - Execution of a process instance with preconfigured actors

The problem with this approach is the lost flexibility, because sticking the process configuration and the involved actors together, makes it more difficult to change the process later on[4]. Additionally, because every actor must store its process configuration in its own internal configuration, a simple replacement of an actor is prevented. In order to replace an actor, the process configuration from the internal configuration has to be moved to the new actor.

1.4.3 Intelligent business objects

Instead of relying on preconfigured actors, the suggested approach in this thesis is to dynamically configure the actors with the business object itself (Figure 8). That way, business objects not only convey just the process instance data but also the process configuration, hence such business objects will be referred to as Intelligent Business Objects or IBOs in short. The disadvantage of using this approach is, however, that more data is passed between the actors, as the process configuration for all involved actors needs to be passed around all the time, including instance data of each actor.

[3] The business objects are tailored for each actor and business process, so that only the minimum amount of data is send between each actor for each process instance.
[4] See also chapter 1.6 - Related Work, as the approach of [9] also has this disadvantage.

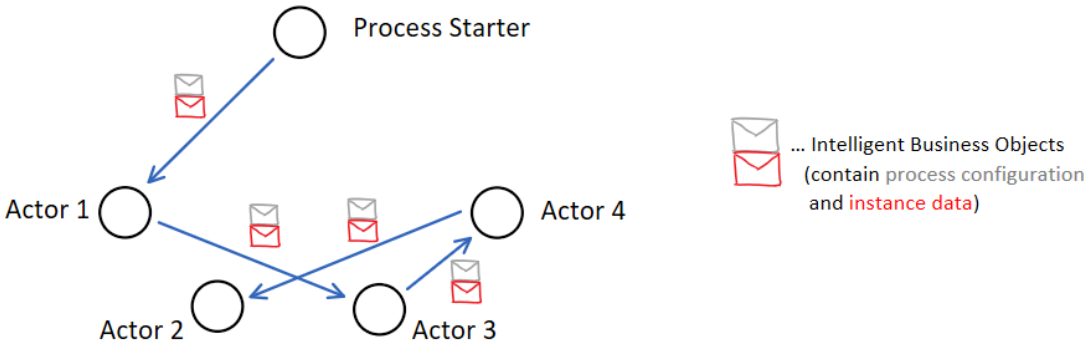

Figure 8 - Using IBOs instead of preconfigured actors

The advantage of using IBOs is the highly increased flexibility. By storing the process configuration in the business object, processes can be modified easily while they are executing by only updating the IBO. Additionally, actors can be replaced more effortless because of a more streamlined internal state of an actor when compared to the previous approach (chapter 1.4.2 - Preconfigured actors).

1.5 Purpose / Methodology

The purpose of this thesis is to show, if the concept of an actor based business processes via intelligent business objects is feasible. This is done by first determining necessary features and a corresponding architecture, which includes these features. Special attention is also required for the reliability of the system, along with mechanisms to handle network failures and other unforeseen errors. But to truly show, if the concept is sound, a prototype has to be developed.

As the development of the prototype might reveal flaws and problems of the architecture, which will not have been considered before, the architecture has to be updated continuously while developing the prototype. The prototype must also include a graphical user interface (GUI), that is used by users to not only interact with running processes, but also to design new processes. The prototype is considered complete, when an example process can be created, executed and interacted with using only the GUI.

1.6 Related Work

The distributed nature of the mail system, which operates without a central control, raises the question, if a process can be executed in practice in a distributed manner to begin with. Previous works by Li, Muthusamy and Jacobson showed, that it is possible [9][5], albeit their approach is similar to the one presented in chapter 1.4.2 - Preconfigured actors and uses BPEL and its block structure[6]. Essentially, this solution

[5] The article uses the term agent, but in this case it can be seen as a synonym for actor. For continuity reasons it is replaced by actor in this thesis.

[6] In detail: The structure of a BPEL model is split up into their individual parts. These individual parts are then translated in a way, that they can be distributed over the actors by configuring them to subscribe to certain events. By triggering a starting event, the first actor invokes its given part of the

is an extended BPEL system, where load balancing is not handled by having multiple BPEL engine instances in a cluster, but more sophisticated by splitting processes over several actors. [9].

Li, Muthusamy and Jacobson's approach acts as a proof of concept for dividing the execution of a process between several actors, but only maps the already proven block structure of BPEL on actors. It also leaves automatic detection and error handling for future work. However, the proposed approach in this thesis is based on a graph or flow based structure, which is more similar to BPMN.

process. When this first actor is finished with its part, it invokes another event, which subsequently triggers the next actor. The communication is done via an enterprise service bus with publish/subscribe and events.

2 Intelligent Business Objects

This chapter explains the ins and outs of Intelligent Business Objects and how they work together to execute business processes. Nevertheless, it does not contain the details of additional needed actors, which are part of the overall architecture and which interfere with the execution of the IBOs, as this would go beyond the scope of this chapter.

2.1 Functional principle

Intelligent Business Objects can be divided into two parts: instructions and instance data (Figure 9). The Instructions are divided into steps, where each step is executed by a single actor. The steps are linked together by the instructions of each step (Figure 10). This way, loops and alternative routes can be created. The instance data is added by the actors, which receive the IBO and are a result of the execution. Instance data can for example depend on user input and is also read by other actors.

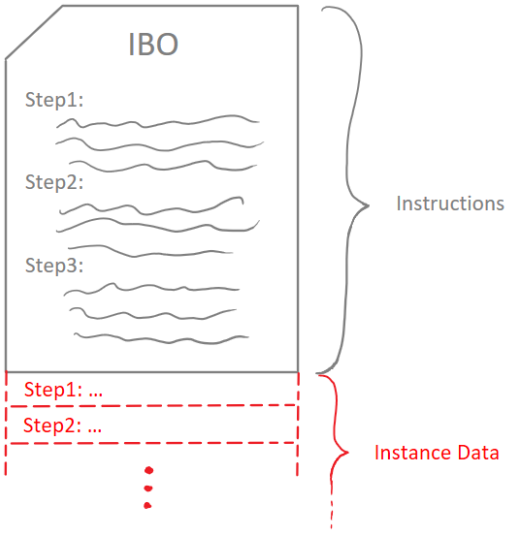

Figure 9 - Basic IBO structure

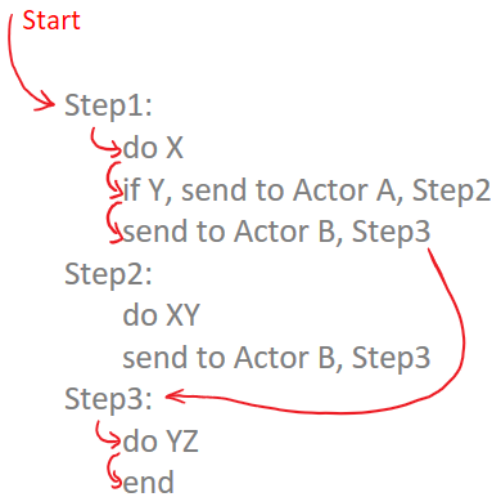

Figure 10 - Instruction linking

Certain characteristics of IBOs are also comparable to conventional software programming: IBOs with its instructions can be seen analogous to classes of object-oriented programming (OOP) and instance data can be seen as class instances.

The instructions and instance data are not completely isolated from each other. Although instructions and instance data are stored separately in the IBO, instructions can link to instance data (Figure 11). By referring to instance data, the execution and the routing can be changed dynamically, without changing the instruction itself.

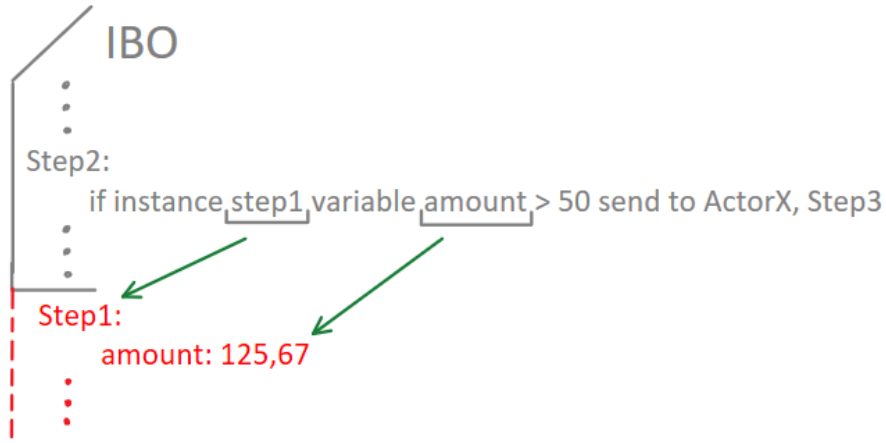

Figure 11 - Instructions linking to instance data

The resulting execution of an IBO in a network of actors is depicted in Figure 12 below. The actors are generally independent of one another, but the IBO instructs them how to communicate between each other. The IBO is therefore acting as the "glue" between the actors.

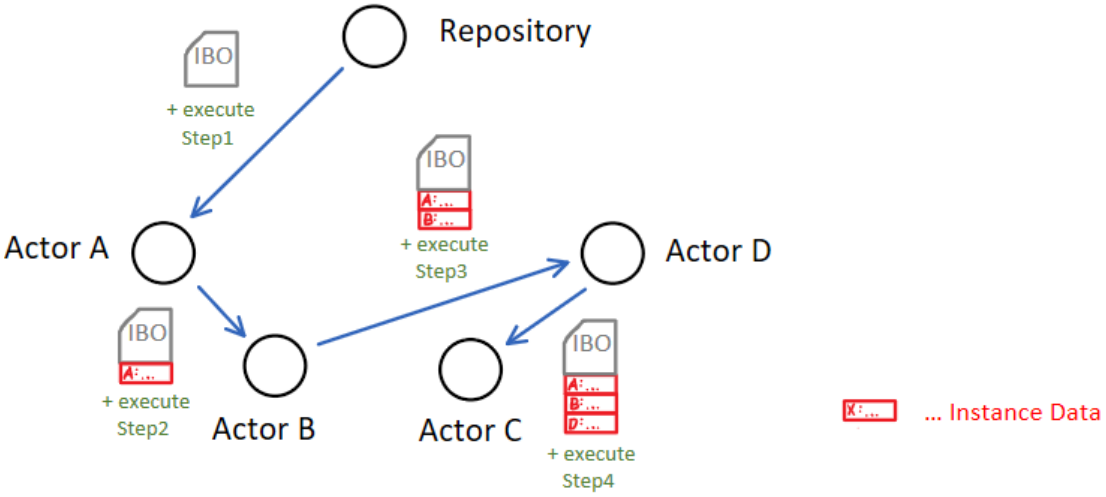

Figure 12 - IBO functional principle

The following sequence of actions are taken by each individual actor which receive an IBO.

1. An actor receives an IBO and which step the actor must execute from the previous actor.
2. The actor executes the given step of the IBO, also taking instance data of previous actors into account.
3. After executing its step, the actor can add its own instance data to the IBO.
4. Before the actor is done, the actor sends the updated IBO to the next actor, also including which step the new actor has to execute. In case there are no more steps, the actor can discard the IBO.

Ordinarily, the instructions of an IBO do not need to be changed throughout the execution of the IBO, because adding instance data to the IBO is enough. But in certain cases, it may be necessary to also change the instructions of the IBO as well. For example, when the process has to be changed, it might become necessary to also update currently executing IBOs and their instructions accordingly.

2.2 Origin

The idea of Intelligent Business Objects is based on routing slips (also called docket, route card, control slip or batch card) from manufacturing (Figure 13). In the production of goods, a routing slip is used to identify the completed steps and the next open steps. More precisely, workers typically get boxes with semi-finished items and routing slips. With the information from the routing slips, workers infer what they have to do with the items in the box. When done, they simply tick of their task and forward the box to the next worker.

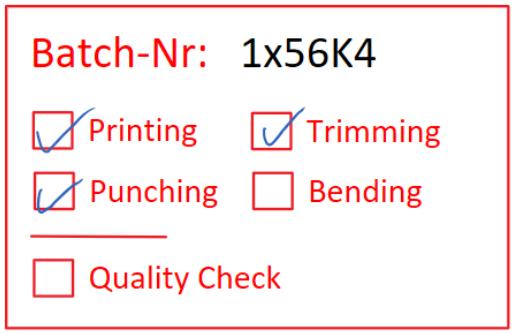

Figure 13 - Example routing slip

An Intelligent Business Object is therefore like an advanced routing slip, that also contains all information what has to be done by the actor and enables a much more sophisticated routing. It would also be possible to use the same IBO principles in manufacturing. Instead of using a routing slip made out of paper, an RFID-tag can be used, which contains more information. For example, the RFID-tag could store machine configurations, so automated machines could be configured automatically.

2.3 IBO data structure

How the IBO is structured in particular is not that important, as long as the involved actors can interpret the given structure. The structure proposed in this chapter should therefore be seen as one possible structure, which is compatible with the system described in the next chapter.

Figure 14 below depicts the suggested data structure. The IBO contains several individual attributes, instructions in the form of a list of steps and instance data in the form of a list of step data.

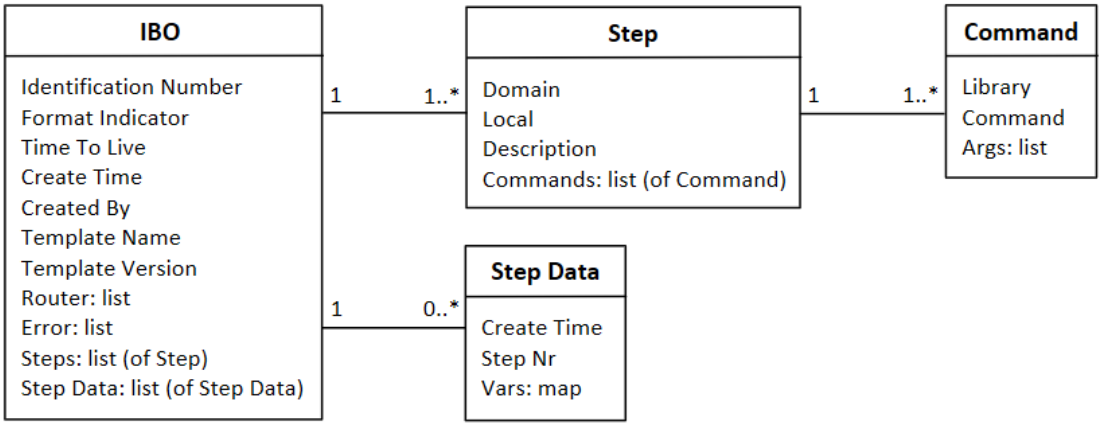

Figure 14 - IBO data structure

2.3.1 IBO - Identification Number

Every IBO needs its own unique *identification number* (*ID*), as this number will be used by the actors as a common identifier. Considering that the system consists of many independent actors, the *Identification number* cannot use just an incrementing number alone, as two actors generating new IBOs would create conflicting *IDs*. Therefore, the identification number should consist of at least two parts[7]: an identifier for the actor, which is generating the IBOs, and a consecutive number, that is increased by one for each generated IBO.

2.3.2 IBO - Format Indicator

The structure of the IBO can change over time, especially when the current structure cannot handle a new problem. Having a *format indicator* simplifies the handling of possible errors when updating the structure, as actors can check which formats they support.

2.3.3 IBO - Time To Live

Processes usually have an average duration until they are finished. But when, for example, there are errors in the instructions, which could lead to an IBO not being able to finish at all, the system would not be able to detect or to get rid of such an IBO. By implementing a *time to live*, such errors can be detected in the system.

There are three possible time to lives variation that can be used. The first way is use an absolute[8] time, that removes the IBO if it Is in the system after the time is reached. The second way is to rely on the number of hops[9], that is increased for each passed actor, so that when the maximum number of hops is reached, the IBO is removed

[7] The prototype implementation consists of an additional third part, a consecutive number for each start/restart of the actor. By using an additional number for each start/restart, the actor does not need to persist the current running *ID* to disk for every newly generated IBO, but only at the beginning of every restart. This prevents a possible bottleneck on the disk, especially if a remote storage space is used.

[8] A relative time could also be used, when a *create time* is implemented, but then an additional calculation is necessary before maximum *time to live* can be checked.

[9] This is what time to live refers to in the internet protocol [14].

from the system. The third way would be to use both, the absolute time and the number of hops.

2.3.4 IBO - Create Time

For statistics and traceability, the time the IBO was created should also be included. For example, the *create time* is also an important variable that can be used as a filter when searching for past IBOs.

2.3.5 IBO - Created By

For similar reasons like the *create time*, it should also be known who started a new IBO. The field *created by* can either contain an identification of another actor or of a user.

2.3.6 IBO - Template Name, Template Version

IBOs are generally created from the repository actor (also see chapter 3.10 - Repository actor). IBOs, who have the same *template name* and the same *template version* usually[10] also have the same instructions. *Template name* and *template version* therefore enable the comparison of IBOs and a statistical evaluation, like average runtime and the paths that the IBO took throughout the network of actors.

2.3.7 IBO - Router

This field stores the list of routers that are utilised by other actors when forwarding IBOs. Routers are actors, used to log and track IBOs and are needed because of the chosen implementation of chapter 3.5 - Logging. The field *router* is implemented as a list of routers to increase the fault tolerance, otherwise the router actor would become a single point of failure.

2.3.8 IBO - Error

Similar to the field *router*, it is a list of actors, which other actors use to send the IBO to when they detect an error in the IBO. For instance, when an actor fails to execute an IBO, it tries to send the IBO to one of the listed *error* actors (also see chapter 3.12 - Error handling).

2.3.9 IBO - Steps

The *steps* field consists of several independent steps. Each step is executed on one actor and each step is also a composite structure made out of several fields. As it contains the execution instructions, an IBO must have at least one step.

[10] "Usually" because the instructions can be changed later on by other actors.

2.3.10 IBO - Step Data

The field *step data* is a list of data that contains instance data. Each actor adds its own instance data at the end of the list[11] after every executed step. In contrast to the field *steps*, the field step data is empty in the beginning of the execution.

2.3.11 Step - Domain

This field determines the destination actor. It is used by actors to check if the step they should execute is also meant to be executed by them. The purpose of this field is to prevent actors from executing steps they cannot execute, before they try to execute the IBO.

2.3.12 Step - Local

The IBO's destination is not only limited by the *domain*, but can further be restricted by the *local* field. For example, the box actor (also see chapter 3.7 - xActors) uses this field to determine how it should sort the IBOs and which users have access to it.

2.3.13 Step - Description

The step can also contain a *description*, which is used purely as an information. It is not necessary to use, but it is very helpful when designing IBO templates and when the IBO has to be edited later on.

2.3.14 Step - Commands

The field *commands* is a list of commands, which are the instructions that the actors execute. As every actor needs at least one instructions to send the IBO to the next actor or to end the IBO execution, the field commands cannot be empty. Additionally, the first command of the list of commands can be used for the initialisation of the actor. This necessary when the actor does not execute the IBO right away, but instead uses the initialisation for setting up the actor for further interactions. For instance, the box actor uses the initialisation to set up an external interface, which is then used to display an input form on a web client (also see chapter 3.7 - xActors). When an initialisation is used, the field *commands* must contain at least two commands.

2.3.15 Command - Library, Command, Args

In order to execute a single command, the actor needs to know which command should be executed (=field *command*), which library the command is from (=field *library,* also see chapter 2.4 - Libraries) and which arguments should be used with the command (=field *args*). Before execution, each actor needs to verify if the given library and command is even allowed to be executed on the actor. This check ensures, that actors do not execute arbitrary commands.

[11] The Erlang implementation of the prototype actually adds the instance data to the beginning of the list, as it is quicker to find the latest version of a step's data (each step can execute the same step several times, also creating step data with the same step nr, so the first occurrence in the list is always the latest version).

2.3.16 Step Data - Create Time

The step data structure contains the field *create time* for statistical evaluations. With it, several key figures can be calculated like running times, but it is also possible to use the field for error handling as well. For example, in case of an IBO collision (= same IBO with the same ID in the system but with different step data) the *create time* could be used to resolve the collision.

2.3.17 Step Data - Step Nr, Vars

Actors add their own step data to the IBO. In order to map the step data to a particular step, step data contain a *step nr* field[12]. There can be multiple step data for a single step and actors generally always access the latest version of a particular step. The actual data is stored in the field *vars*, which is implemented as a key-value map. By referring to a particular *step nr* and a key, actors can access the corresponding value.

2.4 Libraries

IBOs contain instructions, which are executed by the actors. For a better management of the available instructions, that actors can execute, they are sorted into libraries. Every actor has to implement at least a basic library, that is used to forward the IBO to the next actor or to end the execution of the particular IBO. Obviously, only libraries that an actor implements can be used in an IBO, and thus, each command in the IBO has to have a corresponding function on the actor (Figure 15). An actor can therefore also be seen as an interpreter, which uses the instructions of the IBO as its source code.

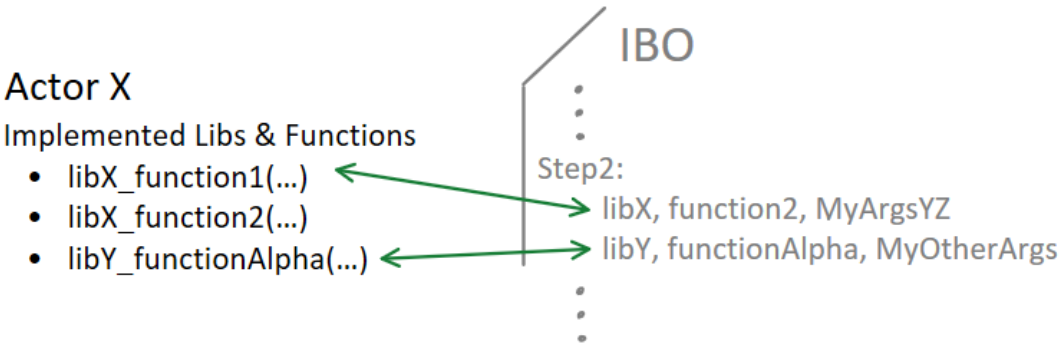

Figure 15 - Link between commands in the IBO and functions of an actor

In order to increase the flexibility, it is also possible to define individual libraries for each actor. This should be done to formalize the interactions between an IBO and an actor and serves as a contract or interface between the two. Libraries thus simplify the process design and the actor creation, because they can be done by two different programmers, which are bound by the same library. However, libraries should always be implemented completely and not partially to avoid “not implemented” errors.

[12] Steps do not have a particular *step nr* assigned to them but instead the *step nr* is simply the index of a step in the list of steps.

2.5 Process step execution

Actors execute an IBO step in a very basic way: they simply try to execute the first command in the list of commands. But the execution does not stop immediately after the first command. Instead, commands call a "next" function, which in turn will execute the next command of the list. This recursive function calling[13] will be repeated until a function returns a value, which indicates if the execution was successful or not.

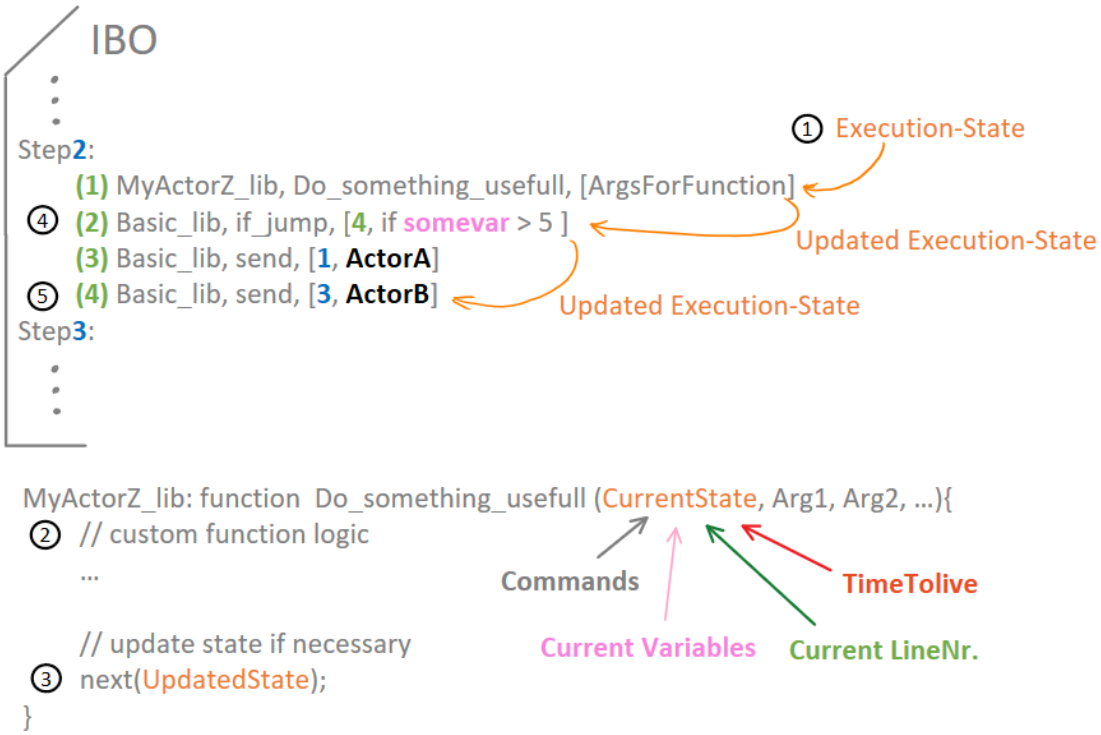

Figure 16 - Execution of a process step

Figure 16 is a detailed depiction of what happens, when an actor executes a single step of an IBO:

1. The actor calls the first function "Do_something_usefull" of the library "MyActorZ_Lib" and passes the *Execution State* as the first parameter to this function.
   The *Execution State* consists of several elements:
   - The step's *Commands*, which have to be executed, including instance data from other steps (grey).
   - *Current Variables*, which contain variables from the current execution of the actor. They can be set by any function that the actor is executing (pink).
   - *Current Line Nr*, which is a pointer to the current step (green).

[13] The erlang implementation uses a tail recursive loop, which prevents the call stack from getting filled up.

- *Time To Live*, a number which indicates the amount of recursive calls to prevent an endless loop (red).

2. The function "Do_something_usefull" executes some custom code. The function can also modify the current state.
3. When the custom code of "Do_something_usefull" is finished, it must call the "next" function and pass the *Execution State* along with it.
   The "next" function increases the *Current Line Nr* and the *Time To Live* by one. If the *Time To Live* reaches a set limit, the next function stops the execution by returning an error, otherwise it calls the next function in *Commands* instead.
4. The basic library contains a function called "if_jump"[14], which manipulates the *Current Line Nr* depending on a condition. This enables the execution to take different branches depending on the provided input.
5. As soon as the function "send" is called, the recursive execution stops. This is because the function "send" does not call "next", but returns a value instead. Based on the returned value, the actor knows, to which actor it has to send the IBO to next and which step nr it has to pass along.

In some cases, actors might require an initialisation, before the actual execution of the step. This can be useful, when the actor cannot execute the IBO straight away but needs to wait for example for additional input. If an initialisation is used can be determined by the function name "init". When this is the case, then the actual execution of the step will start in the second line and not the first one (Figure 16).

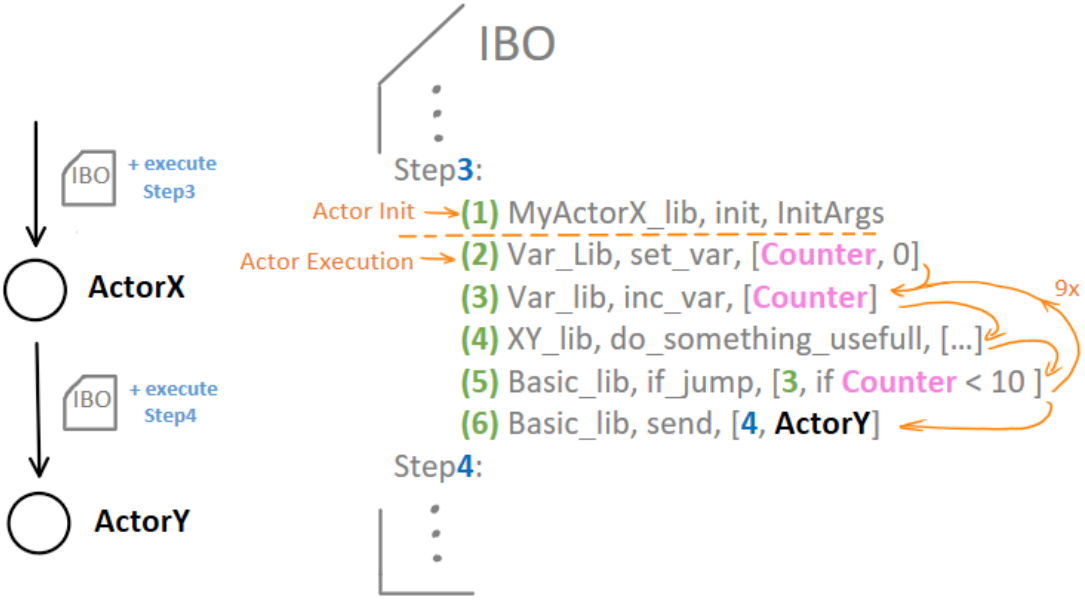

Figure 17 - Actor initialisation and loops

Executing a step with a recursive loop might seem to be counterintuitive compared to the typical block structures used in conventional programming, but it is quite

[14] The function is called "cjump" in the prototype.

simple to implement. By manipulating the execution via conditional jumps[15] (Figure 17, line 5), constructs like if conditions and loops can be created. However, instead of relying on conditional jumps, it would be also possible to create a library, which implements conventional structures like while-loops.

[15] This implementation was inspired by assembly programming.

3 IBO System Architecture

The following chapter describes step by step the necessary parts of the proposed actor based business process system. By referring to the limitations of each step, the system is continually expanded until all parts are outlined, including considerations between different implementation possibilities.

As the system is based on actors, the philosophy of actors is also applied throughout the development. Thus, to resolve a new arising problem, the introduction of a new actor is preferred, instead of increasing the complexity of existing actors.

3.1 Programming language and frameworks

An important decision, that has to be made for the development, is, which programming language and framework should be used. People tend to use what they know best and programmers are not any different. This is also described in the adage: "When you only have a hammer, every problem looks like a nail". But the bigger the task, the better it is to use the right programming language and framework, as more time will be saved later on (and the time saved for not needing to learn a new language/framework disappears). It is therefore advantageous to find the right tool for the given problem, instead of relying on a well-known hammer.

3.1.1 Deciding on a language / framework

Depending on the chosen language and framework, the architecture can change, as certain languages and frameworks limit or favour different implementation possibilities. The more utilities a language and framework provides, the better, as less code has to be written, which saves development time. Another deciding factor is, how simple and straightforward the framework can be expanded, when a necessary feature is not supported by the framework.

A crucial aspect of the IBO system is also how reliable it is, as failures of business software can make or break a company. If a software is not operating as it should, it can have fatal consequences, for instance when airlines are not able to board passengers because of a software problem it forcing the airplanes to stay on the ground. Additionally, the software must also be performant, as unresponsive software also slows down the people working with it and the systems connected to it. It is therefore mandatory for the software to scale up when the load on the system increases, which implies that it needs to be distributable.

3.1.2 Erlang/OTP

Erlang with its Open Telecom Platform (OTP) was given preference over other programming languages and frameworks because of several reasons. Erlang is not just any programming language: it is a functional language that has the actor model build into it:

> *"In practice, Erlang forces you to write actors (processes) that will share no information with other bits of code unless they pass messages to each other. Every communication is explicit, traceable, and safe" [13]*

The OTP then bundles common behaviours together, abstracting code, so that programmers can focus on the actual business logic. Additionally, Erlang/OTP also scales very well and is very robust. The high fault tolerance is achieved by letting parts of the system crash when an error occurs and then handle the crash well, instead of preventing all possible errors, because failures happen naturally. [13]. All these features make Erlang/OTP very suitable to be used with IBOs and has also influenced the decision making process of how later upcoming requirements should be implemented.

In Erlang/OTP, actors are essentially processes which run on a virtual machine called node [13]. When two nodes are connected with each other, then actors can already communicate with each other. If these nodes are running on the same computer or two computers connected via a network is not relevant for the processes running on these nodes, which simplifies the writing of distributed applications.

3.2 Basic System

Simply put, IBOs are self-containing process instances. From a network engineering point of view, they can be seen as network packets "on steroids", as the IBOs, which get passed around by actors, behave like packets in a computer network, but additionally, IBOs also contain execution and routing data.

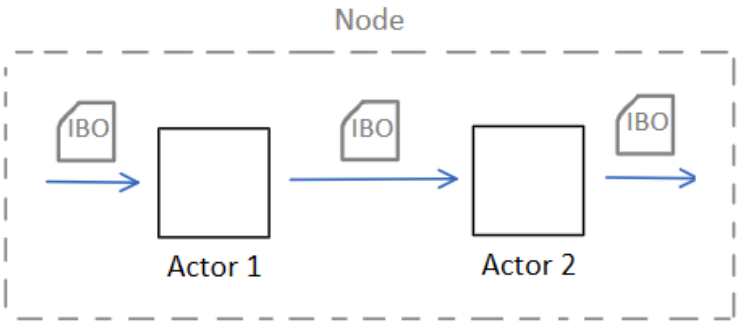

Figure 18 - Most basic system

Using only one node and non-dynamic actors[16], actors should always be available for each other (Figure 18). However, basing the business process system on static actors only would defeat the purpose of creating a business process system on actors to begin with, because only dynamic actors allow the system to be expanded modularly.

3.3 Watchdog actor

The controlled adding and removing of individual actors is done by a separate watchdog actor. This allows not only the dynamic adding and removing of actors at runtime, but also enables a more simplified creation and maintenance of individual actors, as management code is located centrally on each node. Evidently, every node needs to have its own watchdog actor, otherwise a netsplit would render parts, which are unavailable for the watchdog, unmanageable.

[16] Dynamic actors are actors that can be added and removed at runtime, while static actors are only stopped when the parent application stops.

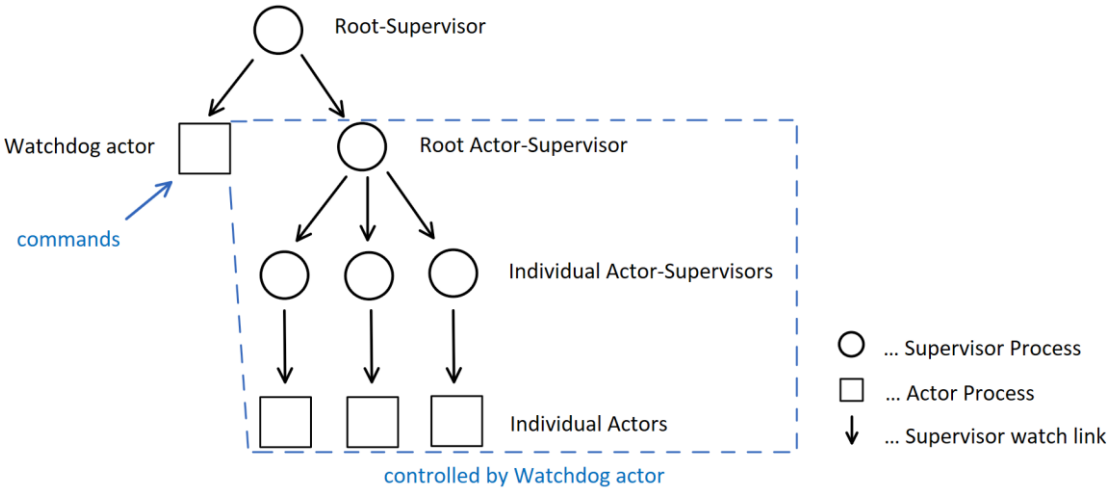

Figure 19 - Supervisor tree for dynamic actor management

Figure 19 depicts the implemented management structure of the prototype implementation. Erlang systems use supervisor trees to create reliable systems, isolating parts of the system and restarting them after crashes, depending on the set restart strategy.

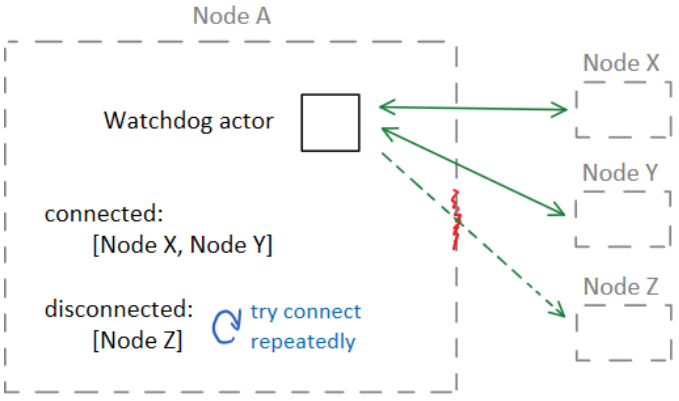

Figure 20 - Node network (re-)connect functionality

By its nature of managing actors on the node, the watchdog actor is also used to connect nodes and to reconnect nodes after network connection problems (Figure 20). This is necessary because Erlang only connects to other nodes automatically when there is an explicit call to the other node, however, the prototype implementation does not call other nodes directly. Instead, actors are accessed by relying on Erlang's build in global module, which calls the right actor only by its global name, regardless of the node the actor is running on.

3.4 Deadletter actor

When adding or removing actors dynamically like in the previous chapter or when actors are spread over several nodes (Figure 21), actor unavailability will occur inevitably and therefore has to be handled.

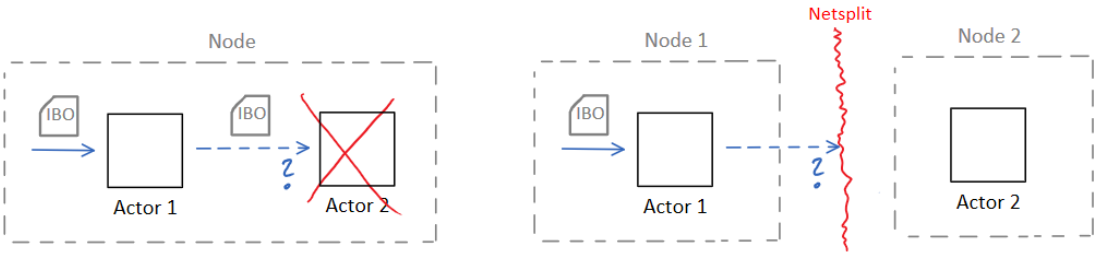

Figure 21 - Issue of unavailable actors

When the destination becomes unavailable, actors cannot finish executing their step. Actors would simply crash, requiring the execution of the same step again when the destination becomes available again. This could create unpleasant side effects, depending on the commands the actor was executing, implying further error handling would be required.

To avoid the increase of complexity per actor, a new actor is introduced, which acts as a stand-in for the unavailable actor, called deadletter actor. Actors, who cannot reach their destination, send their IBO to the deadletter actor instead, which in turn waits until the destination becomes accessible again (Figure 22). Naturally, it is necessary to have a deadletter actor running locally on each node, as network connection problems would again create the aforementioned problem and halt the system.

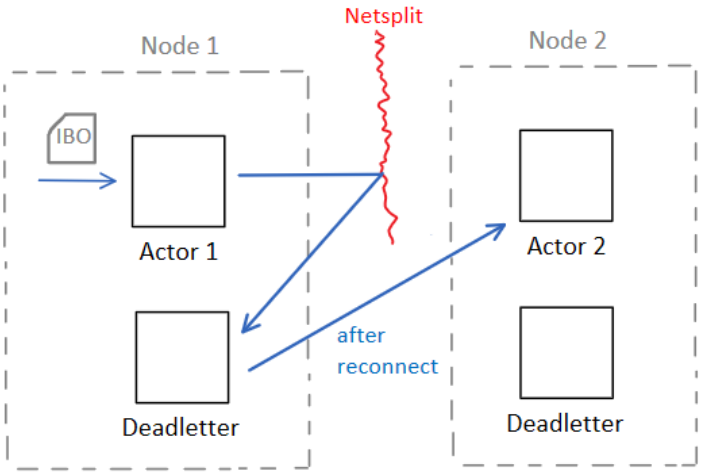

Figure 22 - Deadletter netsplit reconect

By relying on a local deadletter actor, other actors do not need to implement ways to deal with unavailable actors themselves. But it makes the availability of the local deadletter actor crucial for the overall system. To handle this requirement, the deadletter actor is placed above other actors in the supervisor-tree, making it the first actor which is to be started (Figure 23). If the deadletter actor crashes for some reason, all other actors are stopped immediately as well. The Erlang system then tries to restart the deadletter actor, followed by the restart of all other actors.

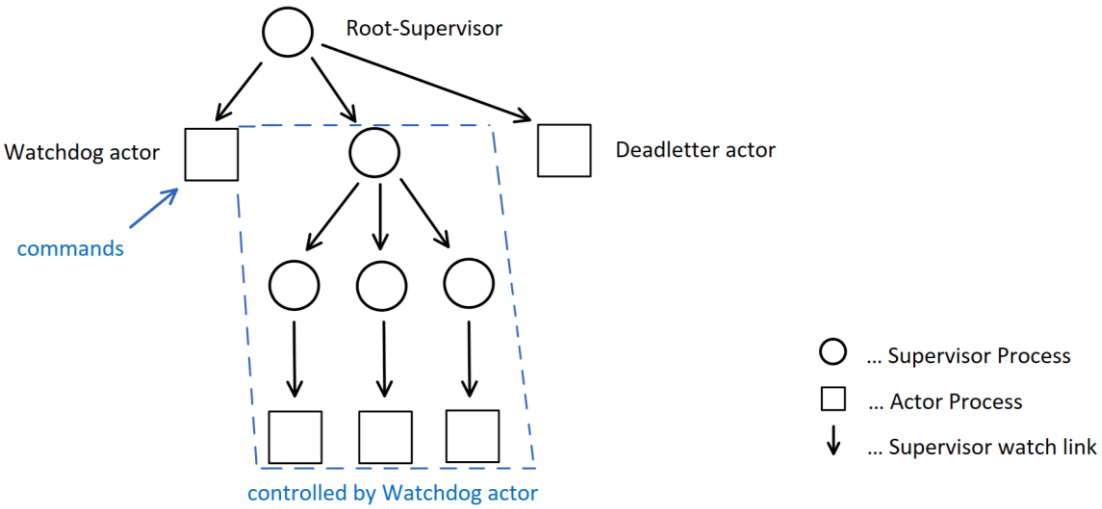

Figure 23 - Deadletter actor in the supervisor tree

Restarting actors can resolve a crash, when the crash issue is related to an unforeseen internal state. Often, crashes occur only when a certain distinct rare order of events occur, making it difficult to debug and even more difficult to anticipate. After restarting, the internal state is "fresh" again, enabling the system to work, until the same distinct rare order of events occurs again. However, restarting does not resolve crashes which happen commonly and constantly. Hence, Erlang stops retrying to start an actor when it crashes several times in a certain amount of time and delegates the restarting procedure to a higher supervisor (until the whole system crashes completely in the worst case).

3.5 Logging

Business processes are crucial for businesses, requiring utmost transparency of the inner workings and the steps in between. As a consequence, traceability or logging of the IBOs is vital when used for business processes and thus needs to be included in the system. Logging can be implemented in different ways, which are more or less practical.

3.5.1 Logging locally

The simplest form is local logging (Figure 24), where each actor logs its own data. But this approach is inefficient and unsuitable for analysis, as all logging data from every single actor needs to be accumulated before further analysis can be done. Nonetheless, local logging should not be discarded completely but more seen as a fallback when other measures are unavailable.

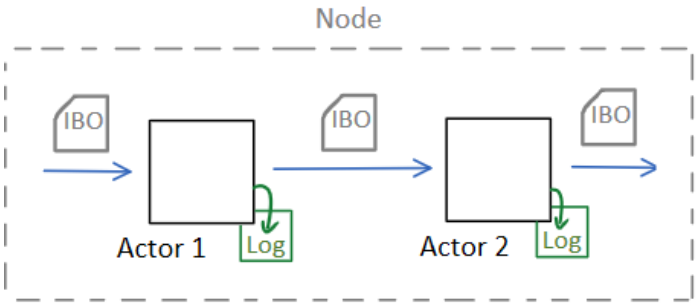

Figure 24 - Logging locally

3.5.2 Logging externally

A more reasonable approach is the use of a logging actor, so actors would send their logging-data to a global logging actor (Figure 25) and the IBO to its new destination, but this would require more programming logic for each actor (as each actor would need to handle the unavailability of the new destination and the logger).

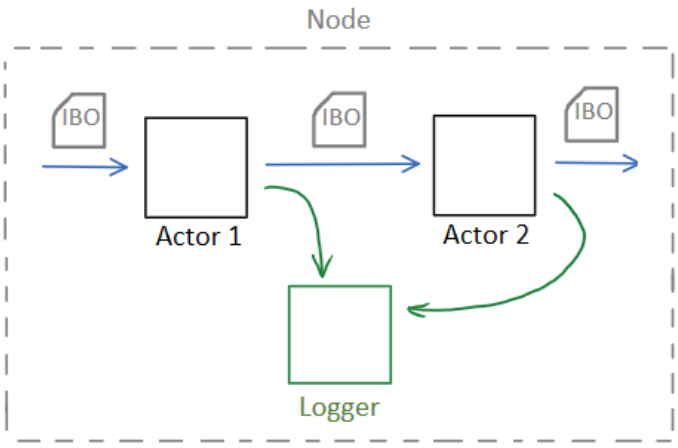

Figure 25 - Logging parallel to the IBO execution

3.5.3 Logging intermediary

The most sensible solution is the use of an intermediary actor, which handles the logging itself and also the forwarding of the IBO to the new destination (Figure 26). This reduces the complexity of the individual actors and adds additional convenient possibilities to the overall systems like filtering certain IBOs on the intermediary actor.

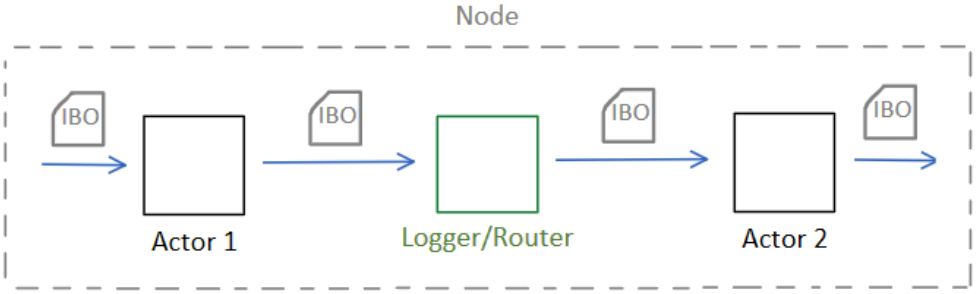

Figure 26 - Logging via router

Adding an intermediary actor also considerably changes the interaction with the deadletter actor, because an unavailable actor would otherwise lead to IBOs bouncing between the deadletter actor and the router, as the deadletter actor sees the available router, but the router then needs to send the IBO back to the deadletter actor (more on that particular problem in the following chapter).

3.6 Router

The primary task of the router is the logging of the IBOs, so that IBOs can be tracked and later analysed. After every step the IBO passes through the router and gets logged. But if the whole IBO would be stored every time, the storage space would be used up unnecessarily. Therefore, only the changes made by the actor need to be stored. As the proposed IBOs have an immutable character[17], only the new step data needs to be stored after every step, as all other data are already persistent.

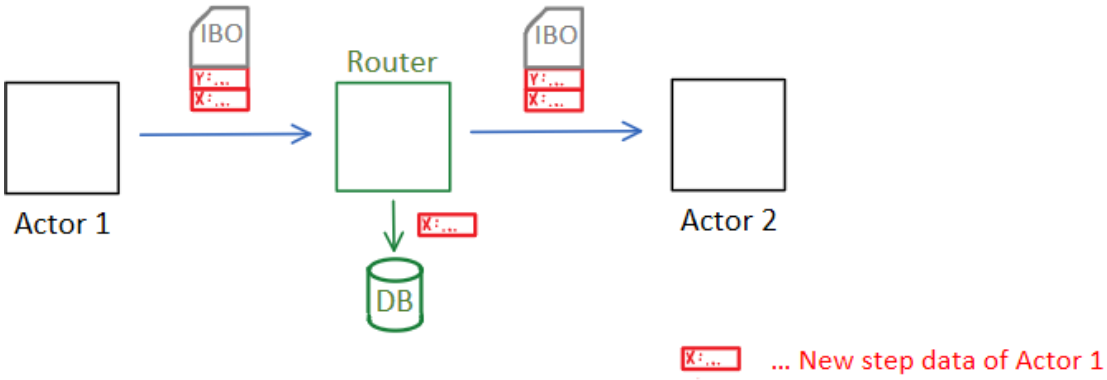

Figure 27 - Logging IBOs

The need to send the IBO to a single router could introduce a new single point of failure, hence several routers can be defined in the IBO. Actors try to send the IBO to the router in the order they are defined in the IBO, if no router can be reached, the packet is send to the deadletter actor instead (Figure 28). Likewise, the router also relies on the deadletter actor when the destination cannot be reached.

[17] Actors generally only update the step data by adding their own step data to the already existing step data.

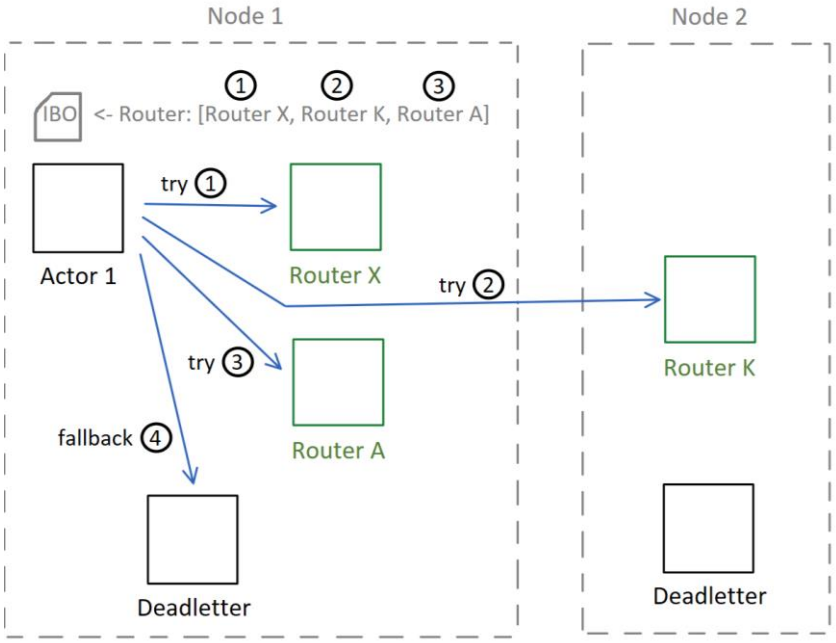

Figure 28 - Send logic for actors

The send logic for a simple actor is straightforward, nonetheless the combined send logic for the router and the deadletter actor is much more sophisticated. Simplified, there are two distinct basic variants that the deadletter actor needs to handle, which are described below.

3.6.1 Deadletter actor receiving an IBO from another actor

When receiving an IBO from an actor, the deadletter actor tries to send the IBO to any of the given routers when one of them is available. However, the deadletter actor does not need to worry if the actual destination of the IBO (the destination that the router tries to forward the IBO to) is available. This is done to log the data as soon as possible, even though the IBO might get send back from the router if the destination is not available. But because there is a distinction between IBOs received from the router and IBOs received from the actor (Figure 29), no race hazard can occur.

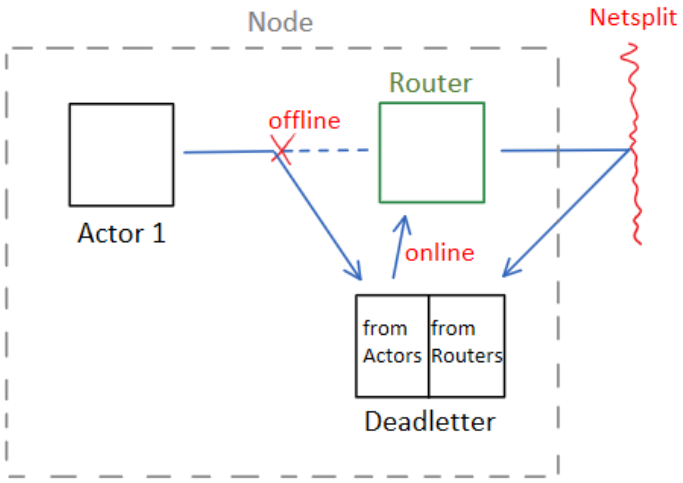

Figure 29 - IBO from actor to deadletter because of an offline Router

If every node would have at least one local router available, then the case for receiving an IBO from an actor could be removed. However, every node would then also need to have access to the database system used for logging (Figure 27), which can be undesirable. When creating special “machine” nodes, which access physical connected machinery, database access could unnecessarily increase the load on such nodes. Additionally, when using other features on the router such as filtering, the more routers there are, the less manageable they become. Furthermore, the current routing logic, where routers are defined in the IBO, would need to be changed and the possibility to define more than one router per node would also disappear.

3.6.2 Deadletter actor receiving an IBO from a router

When receiving an IBO from a router, it is necessary to consider race hazards (or race conditions) and infinite loops, as actors wait for replies from the destination before removing IBOs from the actor’s memory or storage (Figure 30). Hence, the deadletter actor only sends an IBO when a router and the destination is available. This could still lead to the IBO bouncing back and forth, should the destination disconnect after sending the IBO to the router and before the router is able to send the IBO to the destination. To prevent such cases, the router only replies with an ‘ok’ to the deadletter actor when the IBO successfully reached the destination and does not send the IBO back to the deadletter actor (generally, the router replies with ‘ok’ as soon as the router has logged the data).

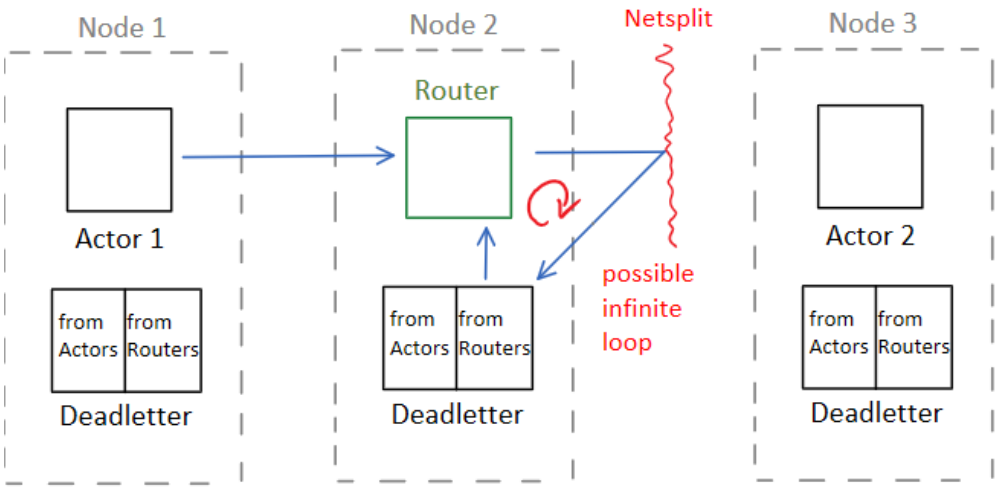

Figure 30 - Possible infinite loop / IBO bouncing between router and deadletter

The deadletter actor could also send IBOs directly to the destination after receiving it from the router. But then it would not be logged, when the IBO is send to the destination and if the IBOs even reached the destination.

3.6.3 Storage considerations

Using only a single router would create a single point of failure, which should be avoided. But having more than one router introduces a new problem, which needs to be considered: how is consistency and availability maintained, when there is a netsplit? The answer is, that consistency and availability cannot be maintained at the

same time when there is a netsplit[18]. If the logging storage would enforce consistency even in the case of a netsplit, then the whole system would halt then, because in order to keep the data consistent, no new data can be written.

Inconsistencies will occur. But the advantage of using IBOs is, that only new data is added at each step[19], and no existing data is overwritten. This can be used to detect and correct inconsistent data, as trying to write already existing data means, that the particular step has already been executed. After logging the collision, the router can simply discard the IBO, as there is already a newer IBO in the system.

3.6.4 Additional possible features

Using a logger as an intermediary device between each actor also enables additional features, which would not be as simple to implement as when a different logging option was used. The following three examples show further possible utility capabilities when using a router.

3.6.4.1 IBO data reduction

When using an intermediary router, it would be possible to reduce the data that is send to each actor, as actors only need to know their own step and the step data of the other steps. This would reduce the amount of data that needs to be transmitted between the actors, but there is also the downside of a higher load on the data storage used by the router, as after every step the next step has to be read from the data storage (Figure 31).

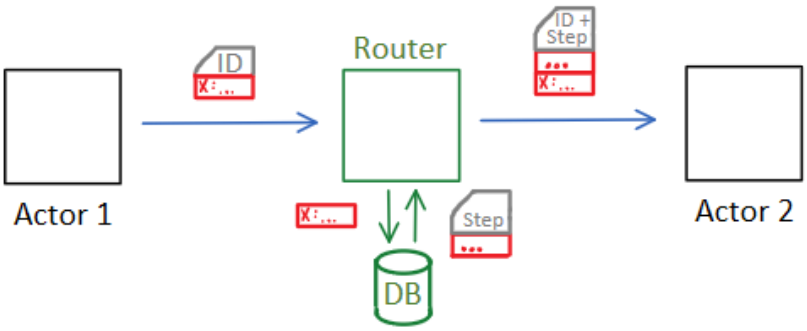

Figure 31 - Necessary changes when reducing IBO data

Commonly, persistent storage is the bottle neck of IT systems. Reducing the data send between actors and routers may therefore be the slower approach, but actual load testing is necessary for a definite assertion. Additionally, the router's logic becomes more complex, but it would also prevent possible manipulations of the IBO by actors.

[18] The CAP theorem states, that you can only have two out of three core attributes when it comes to distributed systems: **C**onsistency, **A**vailability and **P**artition Tolerance. But only the combination CP and AP are useful, as CA implies that the network does not fail. [13]

[19] This is always applicable for storing IBOs. Nonetheless, actors like the update actor can change the whole IBO, but the router gets informed in such cases that the whole IBO has be stored separately (also see 3.12.2 - Consequences of updating the IBO).

3.6.4.2 IBO data filtering

Being placed in between actors, the router can also be used to filter out certain IBOs and prevent the further execution of the IBO after each step. It is also possible to redirect IBOs to an update actor, which updates the content of an IBO depending on other update-rules. This can become necessary if IBOs have been started but need to be changed later on, while they are still executed in the system. For actors, which got removed permanently, but still have active IBOs in the system, "fake" actors have to be defined, which instead of executing the IBO update the IBO instead (an update actor needs to register the old and now unused name and use it for itself).

3.6.4.3 IBO data restoration

As the IBO data is logged constantly between each step, the logged data can be used not only to track, where each IBO has been send to last and to create process statistics for further data analysis, but it is also possible to restore the system's state at a certain point in time. When an actor fails, the logged data can be used to pinpoint the affected IBOs and send them to a new instance of the same actor.

3.7 xActors

The system so far is able to dynamically add and remove actors to compensate for unavailable actors and to log IBOs, but it does not yet describe the actors, who do the actual execution of a step in the IBO. Such actors are called xActors (e**x**ecution Actors) and each xActor expands the capabilities of IBOs. For example, if a process needs to send a SMS notification to a phone, an xActor can be created, which simply allows the sending of SMS. xActors are therefore a good way to integrate all sorts of other systems and services into the IBO system.

Generally, xActors try to execute and finish their step as quickly as possible, so that they can hand it over to the next xActor. But in certain cases, such as the box actor (described below), xActors cannot execute the process right away. This will have further implications later on (for example in 3.13 - Exceptional case handling).

3.7.1 Box actor

The box actor is maybe the most important xActor, because business processes more often than not require further input, which is different for each single IBO (or process instance). But so far, the system is not yet capable of waiting for additional input. Instead of re-inventing the wheel, the requirement can be fulfilled by deriving a new actor from email and its mailboxes, hence the name "box" actor.

Figure 32 shows the functional principle of the actor: the box actor stores IBOs (1) and then simply needs to wait until valid additional input is provided (2), then executes the step (3) with the provided input (2) and sends the IBO to the next actor (4).

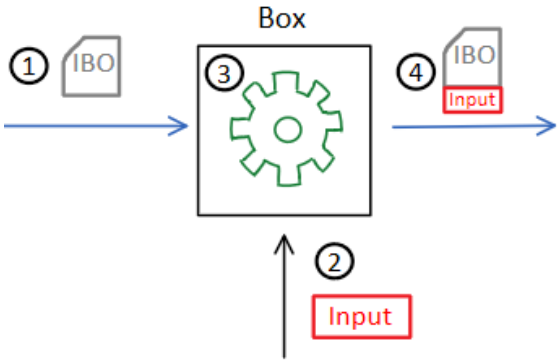

Figure 32 - Box actor functional principle

The box actor behaves similar to an email server (Figure 33). It stores received IBOs (comparable to emails) and sorts them into different groups (which particular group depends on the local variable of the step in the IBO). From a technical point of view, all IBOs are stored in a single table with the ID of the IBO as the key. In a second table previews of the IBOs are maintained, where the group name serves as the key and is used to lookup the IBOs from the main table.

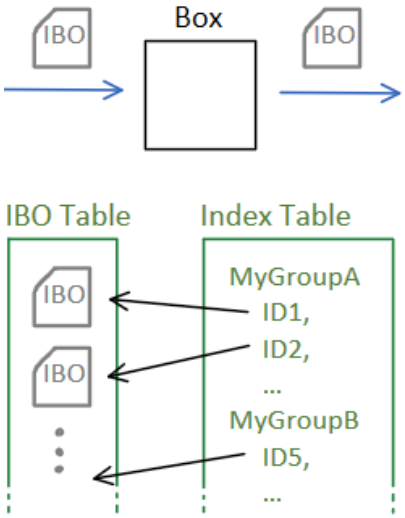

Figure 33 - Box actor datastore implementation

The operating principles of the box actor might seem simple, but there is more to it. In order for the box actor to be usable, the box requires a formal description of the input data's structure or also called schema. Without a schema, it is not only impossible to validate input data, but it is also impossible to know which data input data should be given to begin with. Later on, the schema data will be used to create the necessary input fields (see 3.9 Web actor).

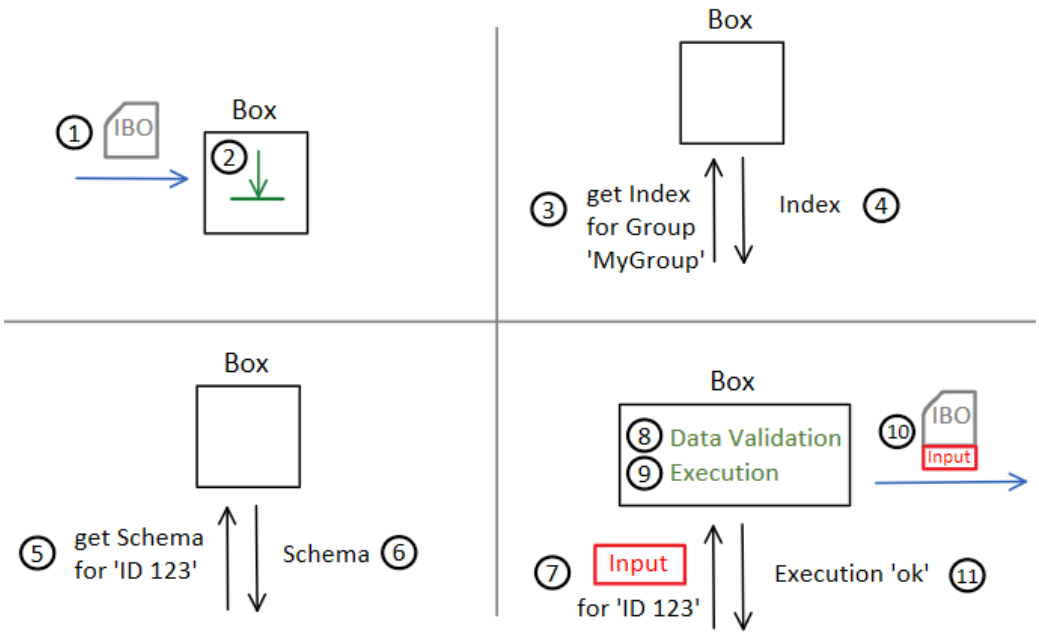

Figure 34 - Box actor interaction

Figure 34 shows the interactions with the box actor necessary until the IBO can finally be executed (9). After receiving (1) and storing (2) of the IBO, the box actor depends on further interactions with other actors. The index lookup (3 and 4) is necessary to know which IBOs are currently available. The result of the index lookup depends on the given group (3) (the IBOs are divided into groups to be able to manage the processes and access rights with users and group-memberships). The index is then used to retrieve a schema for one of the available IBOs (5 and 6). The given schema enables the requesting actor to submit the necessary data (7) but also restricts the data that can be send to the actor (8). After successful validation of the submitted data, the IBO step can finally be executed (9) and send to the next step (10). Subsequently a notification is send that the execution was ok (11). In case the execution was not ok (error in the execution or failed data validation) a different notification is send instead.

3.7.2 Gateway actor

Integrating own services and systems into a unifying business process system increases the value of the whole system. Thus, integrating systems of other companies adds additional value. Gateway actors serve as a link between local business processes and the processes of business partners. Figure 35 depicts an example gateway actor, which communicates with a server of another company. How exactly the gateway actor communicates with the foreign server depends on the available interface. For example, Company B could offer a HTTP/Rest API, so the gateway actor would need to implement this API.

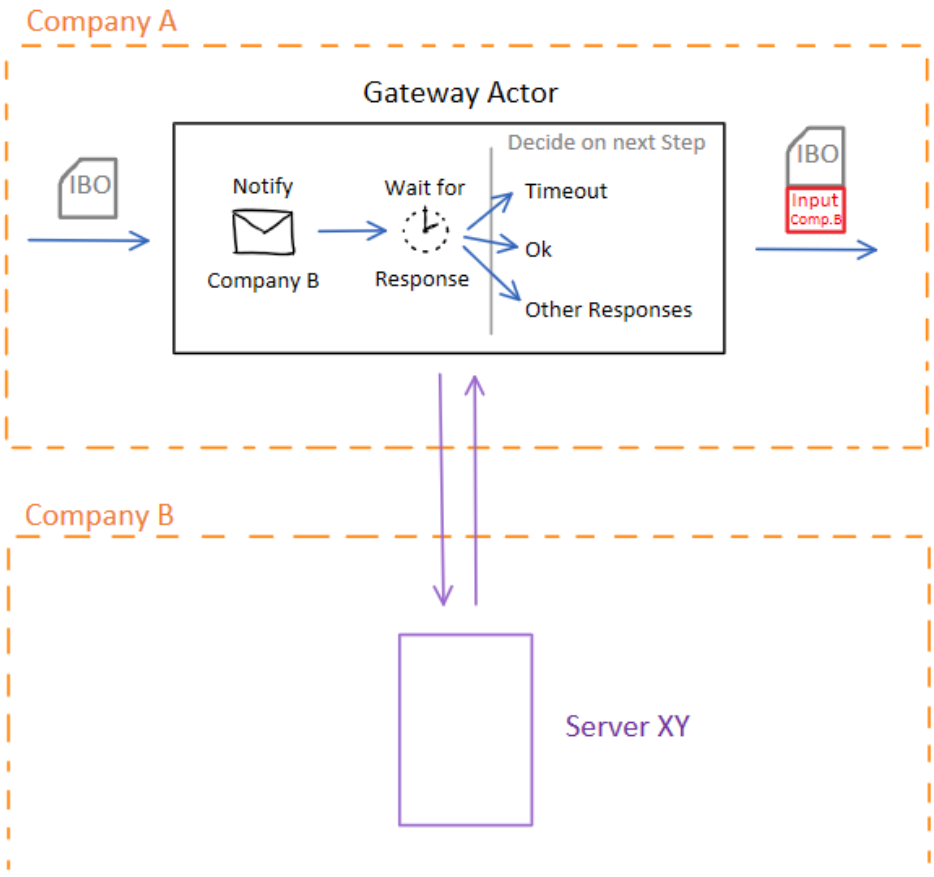

Figure 35 - Example gateway actor

If Company B would also employ the same IBO system, then it would also be possible to simply hand Company B an IBO and wait till it gets returned again. This would simplify the integration, but the negative consequences outweigh the benefits. Not only would companies reveal parts of their internal processes and their actors in use, but it would also expose each system to manipulated IBOs. That is why only the least possible information should be shared instead.

The example gateway actor of Figure 35 can only access an outside interface, but does not provide an interface for outside servers itself. But it is not absolutely necessary to create a dedicated API for outside access, because a similar behaviour can be achieved by simply using a dedicated repository actor, directory actor, box actor and web actor with the correct settings (repository, directory and web actor are described below). Thus, other companies can access the system in a similar way like normal employees, except the ability to define new processes. Hence, these other businesses can not only log into the system with their own username/password and start new processes, but they can also provide input for processes via their own box server. Furthermore, as the web actor contains a web API, other companies could also integrate the IBO system into their own IT architecture.

3.7.3 Email actor

The email actor is an example of the possible capabilities of the IBO system. Everything that is needed to integrate email functionality to the system is to write an email actor and use the newly created actor in a process. Whenever the functionality of the system should be expanded, simply add another actor.

3.8 Directory actor

The functional principle of the box actor requires another actor for the management of users and groups. This directory actor maintains users, groups and the group-membership of users. It also stores encrypted passwords[20] and can therefore be used to authenticate users, but no actor so far enforces any authentication. The box actor for example simply relies on other actors providing group information.

To better integrate the system into an existing IT architecture, it is possible to use the directory actor as a forwarder, which forwards authentication requests to an already existing authentication system (Figure 36), instead of authenticating a user itself. This is a much more user-friendly approach than simple local authentication, because users do not need to remember several different usernames and passwords and can simple use the same username and password for all systems and services.

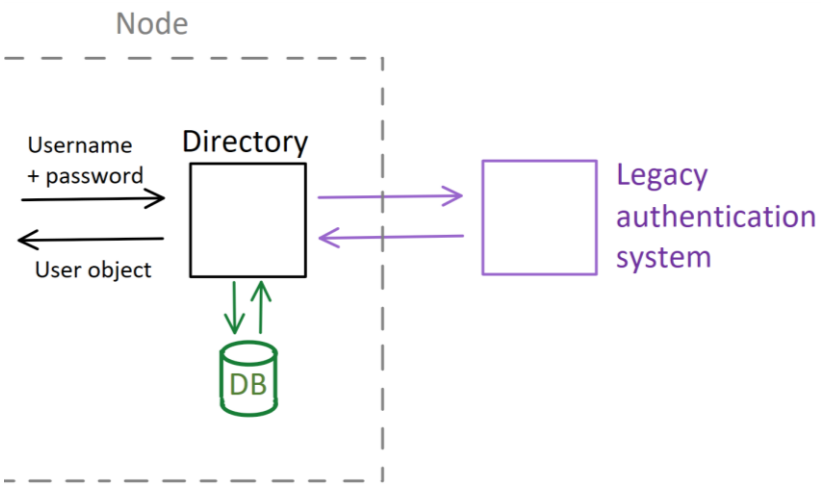

Figure 36 - Directory actor as a forwarder for authentication requests

As other actors do not enforce user authentication, the directory actor only functions as a support actor. If every single actor would require authentication, the directory actor would become a single point of failure for the whole system and also a potential bottle neck. And when it comes to user interactions, it is apparent that some sort of authentication is unavoidable. But user interaction also raises another unresolved issue: how exactly do users access actors? Once again the answer is using another actor.

[20] The prototype uses the PBKDF2 hashing algorithm, simply because there is an easy to use implementation of PBKDF2 in Erlang.

3.9 Web actor

Users need access to actors, but should also be required to authenticate themselves. Additionally, the interaction between actors and users should be kept simple, which is why a graphical user interface (GUI) is essential. All these requirements can be met with new actors.

The new requirements can be divided a client side (or web-client), which is in charge of the GUI, client side validation and communication with the webserver and into the server side, which is in charge of authentication, authorization, session management, data validation and forwarding of requests from the web-client to other actors. The server side is then further divided into the cowboy web-server, which turns HTTP(S) requests into erlang function calls and the actual web actor[21], which configures the web-server and handles session management. This additional division on the server side is necessary to properly include the cowboy web-server as an actor into the overall system.

3.9.1 Authentication

Users have to login before they can access other actors in the system (Figure 37). In order to authenticate, the user sends username + password via the web-client to the web-server, which turns HTTP(S) requests into a format usable by Erlang/OTP. The web-server then tries to retrieve the user object for the given username and password. The directory actor does not simply hand out the user object, but tries to authenticate the user with the given data. If successful, the directory actor, removes the hashed password from the user object and returns it to the web-server.

The web-server then forwards the user object to the web actor, which stores the user object temporarily in memory for future requests. The web actor subsequently replies with the created SessionId. With the user object and the SessionId the web-server can finally inform the web-client, that the authentication was successful. This is done by sending a http-reply to the web-client, which contains the user object of the now logged in user and also sets a http-only cookie with the SessionId on the web-client.

[21] The watchdog actor does not supervise the cowboy web-server, only the actual web actor. The cowboy web-server is a separate erlang application, which is set up by the web actor at its start. Thus, when a web actor is added to the system via the watchdog actor, a new cowboy web-server instance is started. After the start, the web actor is used for session management, until the web actor is removed from the system, which also stops the previously started cowboy web-server instance.

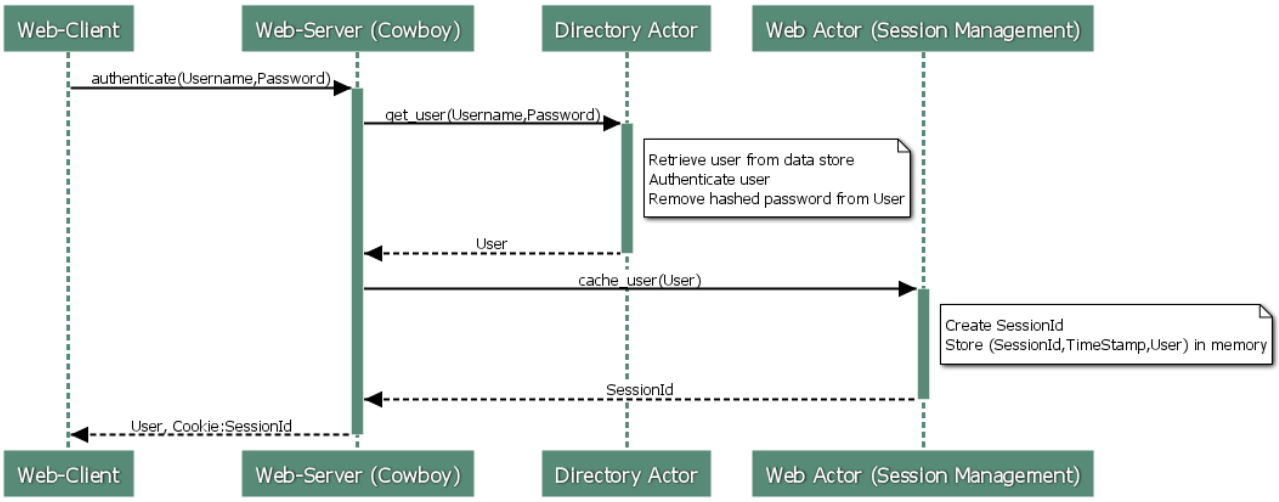

Figure 37 - User authentication (requests to local data stores/in memory stores omitted for simplification reasons)

For all subsequent requests, the web-client adds its SessionId cookie to the request, which enables the web-server to authenticate the request without relying on the directory actor[22]. It also speeds up the authentication process, because the password hashing algorithm on the directory actor is skipped and only a lookup of the session is needed. Using a SessionId instead of sending username and password with every requests also prevents users from being logged in on several computers at the same time, when only one session is allowed to be stored per user.

3.9.2 Accessing actors

When actors are accessed via the web-client, the request has to be authenticated via the given SessionId. The request is authenticated when the SessionId is found in the local storage and has not yet expired. But before forwarding the request, the request itself has to be authorised as well. The authorisation process heavily depends on the actor, that the user wants to access and is therefore not described in this section.

In the figure below (Figure 38) the retrieval of box indices is used as an example to show, how the system handles normal requests from the web-client. When the web-server receives a request to get boxindices from the box actor, the web-server checks the given SessionId by trying to retrieve the user object from the web actor. When the web-server can retrieve the user object from the web actor, the request is authenticated, but not yet authorised. But contrary to most other requests, accessing the boxindices does not require an additional authorisation step, because users can only access their own indices, which are determined by the groups stored in the user object. Thus, users cannot even request indices they do not have access to. Therefore, the web-server only needs to request the boxindices from the box actor for the user's groups and can simply forward them to the web-client.

[22]see Figure 38 for an example that shows how the SessionId is used for authentication.

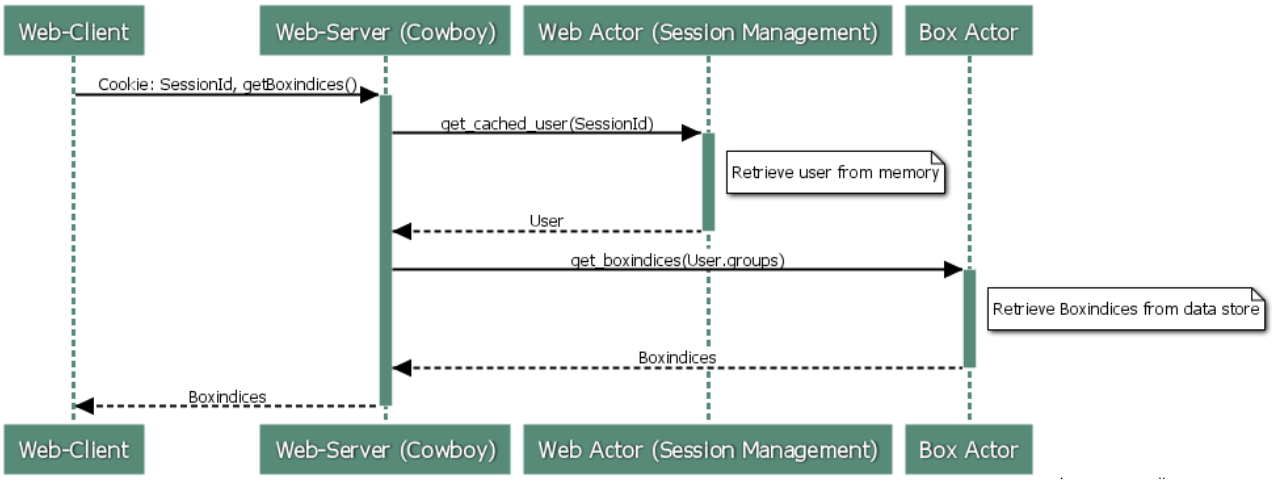

Figure 38 - Accessing box-indices via web-client (requests to local data stores/in memory stores omitted for simplification reasons)

In the Erlang implementation, the web-server is further divided into two parts. The common part contains authentication code for the web-client such as checking the availability of the SessionId, as each request has to be authenticated. The custom part contains code tailored for each individual actor that is accessed by the web-client, which calls the individual actors, but also contains code for authorisation, data validation and data conversion.

In the system so far, users can interact with IBOs and the system is also protected from random, unwanted and potential malicious input by users. But users cannot start new IBOs yet. Once again the solution is to create another actor.

3.10 Repository actor

The repository actor is a comparably simple actor. Its main task is to store IBO templates and to create new IBOs from the templates. But it also requires functionality to list available templates, restrictions to limit which user can execute which template (as certain processes might require access control) and the management of different versions of a template.

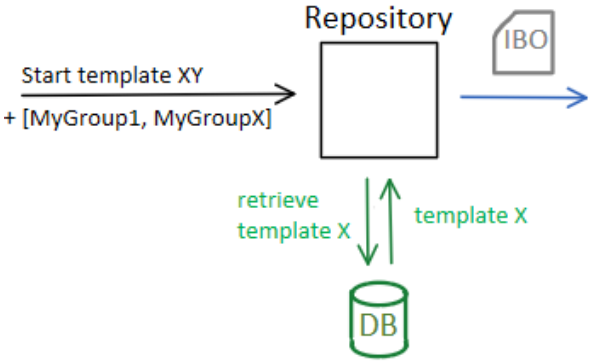

Figure 39 - Repo actor basic principle

When a new IBO should be started, a request is send to the repo actor, which contains the template name and groups that the requestor is a member of (Figure 39). The repository then tries to access the template and checks the given groups with the allowed groups, which are stored in the template. If any of the given groups matches a group from the template, the request is allowed and as a consequence the IBO is created (Figure 40).

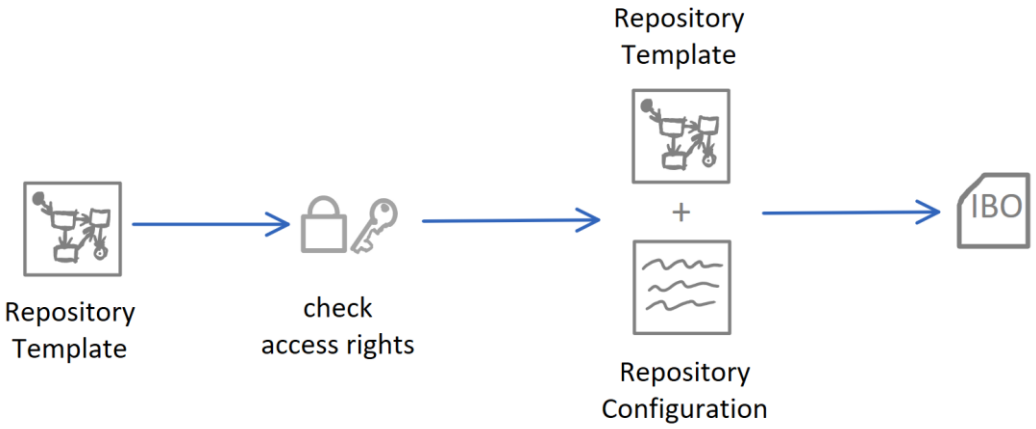

Figure 40 - Repo actor IBO creation

In other words, when generating the IBO, the repo template is expanded with information from the repo configuration. The repo configuration not only contains a list of routers, that are used by the IBO, but it also the necessary information to create a collision free consecutive ID.

It is important that the ID is truly unique, otherwise errors will occur in other parts of the system. Therefore, the currently implemented ID consists of three parts. As there can be more than one repository, the first part of the ID is the repository's name. This should prevent collisions between several repositories. The second part of the ID is the restart counter, which gets increased by one at every restart, while the third part of the ID is a running ID, which gets increased by one for each created ID. This is done to limit the amount of write-calls to the database, as every single created ID would require an update to the persistent storage.

As templates are the basis of the IBO creation, it is necessary to protect the templates against accidental negative changes. Thus, every single template is checked for potential errors before a new template can be stored. Additionally, templates on the repository are never deleted, but instead old templates are moved to a separate table, from which they can be restored.

With the repository actor, users can finally start IBOs on their own, but there is yet another important part of the system missing: the creation of templates by users. Because so far templates have to be defined in Erlang code, which might be enough for testing purposes, but not for any productive use.

3.11 Template creator

Creating a GUI for the template creation is simple and straightforward in theory, but requires substantial programming work. This is caused not only by the different technologies in use (Html/CSS/JS at the client side and Erlang on the server side) which require data conversions in between, but also because creating a responsive GUI involves writing code for the appearance, input forms and validation and additional effort for user friendliness.

The template creator is implemented by leveraging the already existing web server and client to include the loading and storing of repo templates, so the overall system architecture does not change. Essentially, the graphical representation of a template is nothing more than the template from the repository actor together with positional data (X and Y positions) of the steps and the start- and endpoint. Hence, a GUI variable is added to the repo template. The reason for adding a separate GUI variable instead of adding GUI information to the particular steps in the template is because the internal logic of the repository actor does not need to be changed and templates created by the GUI and templates created in Erlang code can be used more easily next to each other.

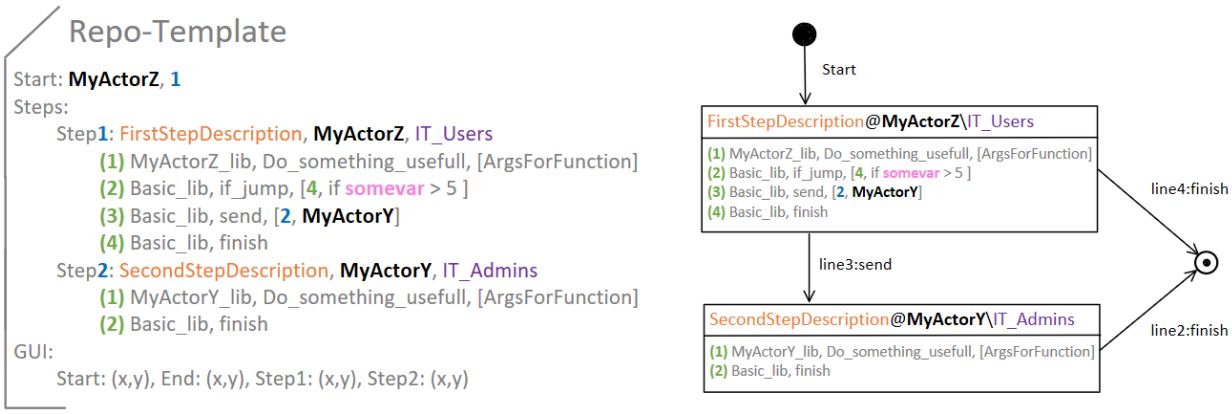

Figure 41 - Repository template and its graphical representation in the template creator

Figure 41 shows an exemplary repo template. As it can be seen in the figure, the graphical representation reflects the underlying structure of the repository template. Each box in the graphical representation stands for a single step executed on an actor. The arrows, which connect the boxes, represent the possible transitions between the steps ("possible" because the used transition depends on the input data). The transition itself are the send and finish functions of the basic library.

Simply put, the graphical representation is nothing more but a graphical editor for the templates. For this reason, the representation does not limit the available functions and it is also very simple to generate a graphical representation of templates, which were only defined in erlang code. The disadvantage is, that it is not a standardised representation of a business process.

3.12 Error handling

In spite of all the described mechanisms to keep the system working, there can be unforeseen cases that actors cannot handle on their own. For such cases, another actor is introduced into the system, called error actor. The error actor is not called directly by other actors, but receives its IBOs through a router (Figure 42), which is done for logging purposes.

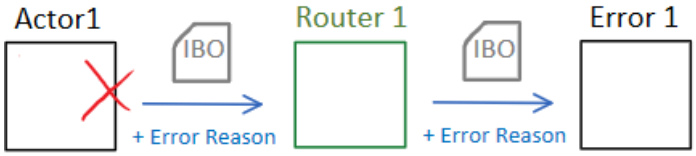

Figure 42 - Actor accessing the error actor

As error actors can be unavailable like any other actor, the deadletter actor as well as the router have to be updated to handle the unavailability of error actors. The changes to these actors are, however, limited, as the previous considerations of how IBOs have to be send through the router apply here as well.

3.12.1 Automatic error handling

The power of the error actor is, that it can update erroneous IBOs automatically and inject the corrected IBOs into the system again (Figure 43), which improves the overall reliability of the system. After storing an IBO, the error actor can check, if it can resolve the issue automatically. This is done by applying available search filters on the IBO. When a filter returns true, the update function connected to the filter is applied, which updates the IBO. The updated IBO is then send back to the actor (via the router), which reported the erroneous IBO at the beginning.

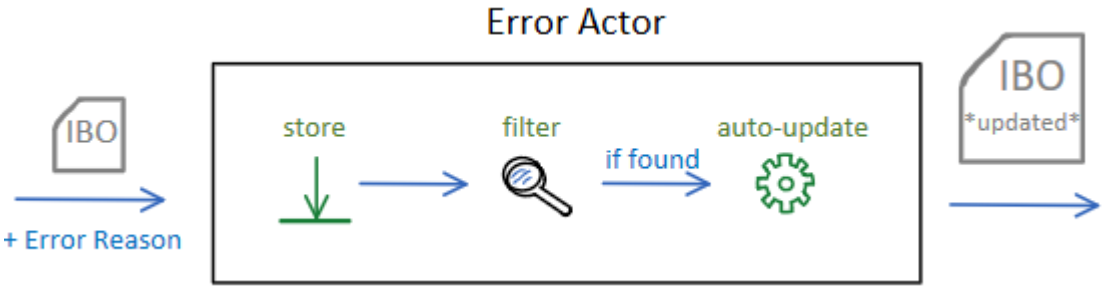

Figure 43 - Automated IBO error fixing

However, there is a pitfall to this approach: when the update to the IBO ineffective and produces the same error again, the same ineffective update would be applied again, which would lead to an endless loop. Because of that, the error actor needs to add its own step-data with information about the update process to the IBO, which allows for the detection of IBOs that have already been updated, but which failed to resolve the error.

3.12.2 Consequences of updating the IBO

The router usually only needs to store updated step data in its data store (see chapter 3.6 - Router). But when the IBO itself gets updated, the router must be informed, that more than usual has been updated. This can be done by changing the template variable, as the router can then notice easily, that that particular template has not been stored yet and is able to log all the changes done to the IBO.

3.13 Exceptional case handling[23]

So far the system is already capable of the execution of processes, including the design of processes, error handling, user interaction and logging. But nonetheless, there can be exceptional cases, where the normal execution path needs to be stopped and an alternative path needs to be executed instead. Figure 1 shows such a case, where a customer calls to cancel an order. When the system can handle such cases, the next steps are not executed and money and time can be saved in addition to a better customer satisfaction.

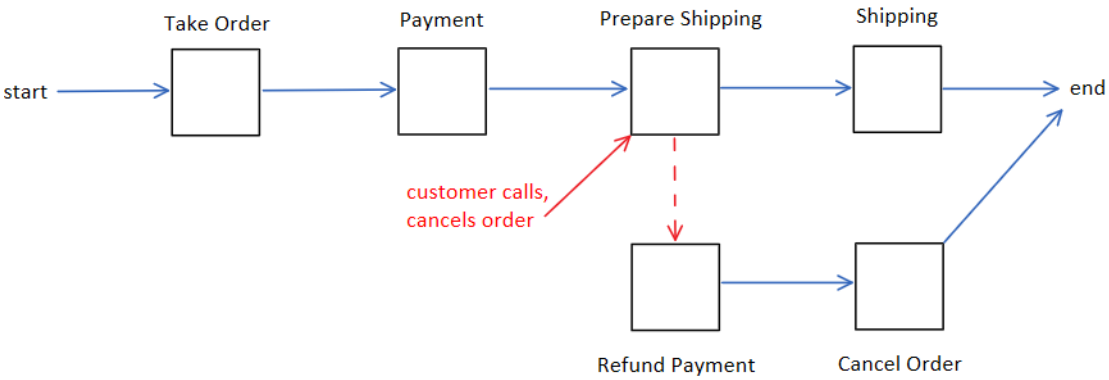

Figure 44 - Exceptional case example

The important points to consider when implementing exceptional cases are:

- At which steps are exceptions allowed and when can they be triggered in the step?
- Who can trigger the exceptional case?
- What has to be done when the exception is triggered?

3.13.1 Suggested implementation

First of all, it does not make much sense to implement exceptional case handling for all xActors. This has to do with the fact that xActors try to execute their process step as quickly as possible. So it is not practical to implement exceptional case handling when an actor executes a step in a couple of milliseconds.

The actor that is most likely to profit from exceptional case handling is the box actor, as the actor needs to wait for additional input before starting its execution. As long

[23] The presented functionality could also be called exception handling, but exceptional case handling is a better description, because the term exception handling has a different meaning in software programming, where exception is very similar to error.

as the actor is waiting for additional input, the box actor can be informed to change the current step. Therefore, the box actor needs to be extended to allow interrupts. When the box actor receives the interrupt, it removes the particular IBO from its local storage and sends it to the actor defined in the interrupt.

Of course, the IBO needs to be updated accordingly as well to include interrupts. This can be done by adding an additional field to the field steps (see chapter 2.3.9 - IBO - Steps), which holds a list of possible interrupts. Each interrupt then contains a name or description of the interrupt, a field to identify, who can trigger the exception and the new step that needs to be executed instead.

3.13.2 Additional note

The same or at least similar behaviour could also be achieved by adding additional steps, but this would make the process more complex and less transparent. Using exceptions is therefore a more elegant solution as shown in Figure 45.

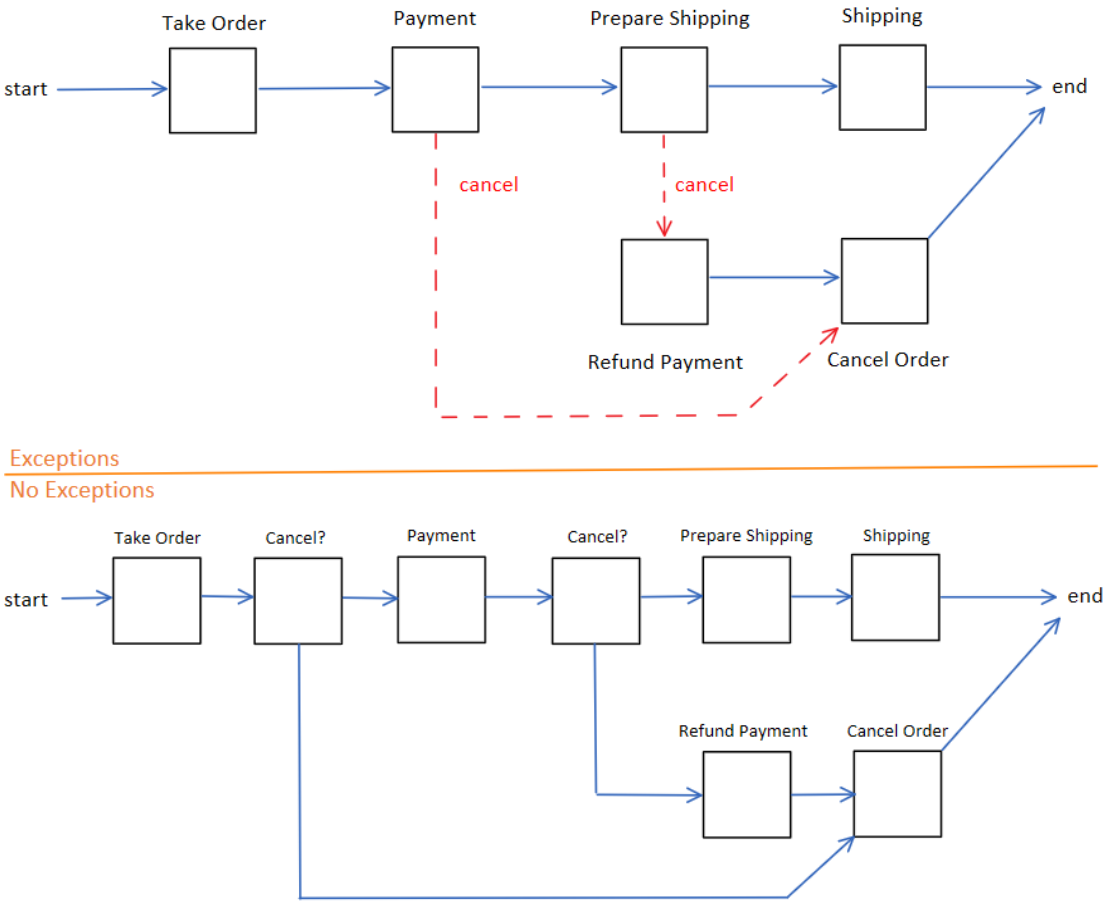

Figure 45 - Exceptional case handling vs. regular process implementation

An additional benefit of using exceptional case handling, in contrast to modelling the same behaviour with additional steps, is the atomic nature of exceptional case handling. Either the step is executed, or the exceptional step is used instead. When the step Payment is reached in Figure 45 (No Exceptions), then the Payment step has

to be processed, the step cannot be stopped any more. But when using exceptions, as long as the Payment has not been executed, it can still be cancelled.

3.14 Update actor

The exceptional case handling is a convenient feature when the exceptional cases can already be predicted in advance. More problematic are the cases, which occur, but that have not been predicted. For such cases, the whole process needs to be updated. Following the actor philosophy, the solution is to add an update actor, that can update the whole IBO[24], including its step data. In other words, the update actor is an editor for IBOs.

The update actor on its own is quite limited. In order to utilize the update actor to its full potential, additional changes need to be made to other parts of the IBO system. The following three exemplary changes help to unlock the update actor's capabilities.

3.14.1 Extending exceptional case handling

When an actor has exceptional case handling implemented to redirect IBOs to alternate execution path, the exceptional case handling can be extended to also redirect the IBO to the update actor. That way, also unforeseen exceptions can be handled by updating the IBO itself. But simply updating the IBO is only part of the solution, the template for the IBO should then also be updated to handle the newly occurred exception.

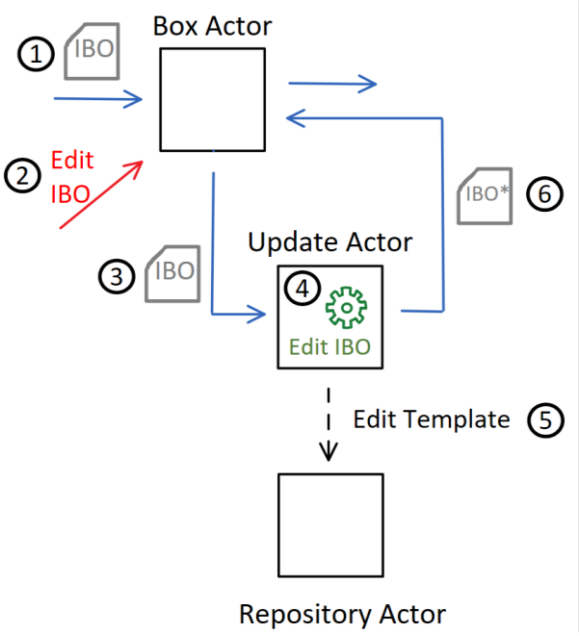

Figure 46 - Extending exceptional case handling to include the update actor (representation omits router actor)

Figure 46 depicts the newly added feature in the architecture. After the box actor received the IBO (1) the box actor can be instructed to send the IBO to the update

[24] Also see chapter 3.12.2 - Consequences of updating the IBO, because the same consequences apply for the update actor as well.

actor (2 and 3). In the update actor, the IBO can be edited (4). If necessary, the changes made to the IBO can be replicated to the template stored in the repo actor (5). The last step is to send the now edited IBO to the original actor (6), but if necessary, it is also possible to define a new actor as the destination of the edited IBO.

3.14.2 Extending the error actor

The error actor so far was only described to automatically fix erroneous IBOs by applying update functions automatically. With the existence of the update actor, erroneous IBOs can also be fixed manually by redirecting them to the update actor.

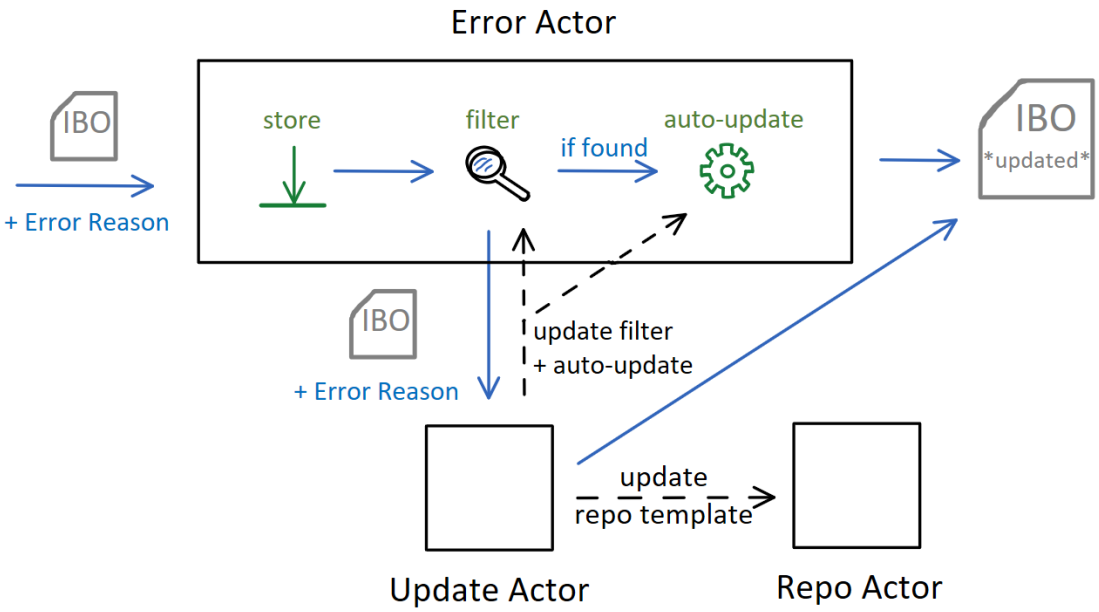

Figure 47 - Extending the error actor to include manual IBO updates (representation omits router actor)

Similar to chapter 3.14.1 - Extending exceptional case handling, the update actor should also help with updating the error actor to automatically add a new filter rule, which fixes IBOs with the same error automatically. Evidently, the IBO template should also be updated, so that IBOs do not even get send to the error actor to begin with.

3.14.3 Extending the template creator

Having an update actor also enables the execution of incomplete IBOs. This can be achieved by defining an IBO template like before, but instead of completely defining the whole process, the IBO is send to the update actor directly in the process. When the IBO arrives at the update actor, a user only needs to define the next steps and can already inject the IBO into the system again (Figure 48). Processes can thus be modelled “on the fly”.

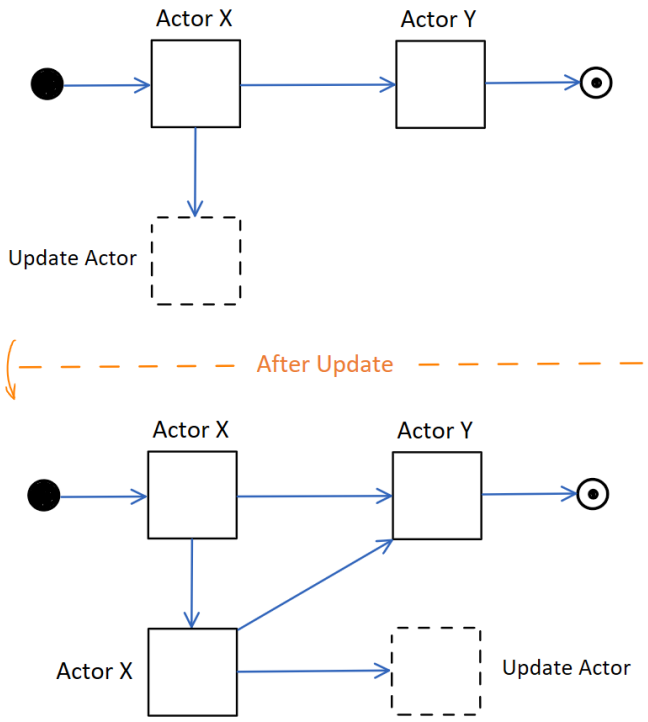

Figure 48 - Incompletely defined IBOs

By using the update actor as a placeholder, the IBO system can be leveraged to create ad hoc processes. This high degree of flexibility is a further indicator of the potential of using IBOs for business process execution.

3.15 Resulting Architecture

The above described actors together form the complete system. Each actor (except the deadletter actor and the watchdog actor) has a uniquely identifiable name, which is used to locate the actor in a system of several nodes. The actors only communicate by their names, so the physical location is not known to the actors themselves. This results in a different logical architecture perceived by the actors in contrast to the actual physical architecture.

3.15.1 Logical Architecture

Figure 49 depicts the resulting architecture and the interactions between the actors. As it can be seen, each actor has defined ways of communication and also different dependencies. Two actors stick out, as they have the most connection to other actors: the router and the web server. The router is in charge of logging all IBO activity, which is why no IBO is send directly between actors but only via the router. The web server has several connections, as it forwards messages from outside via http(s) to other actors. Additionally, the web server also acts as a means of access control to the actor network, as it forces authentication and also checks the authorisation so that other actors can rely on the validity of the send messages.

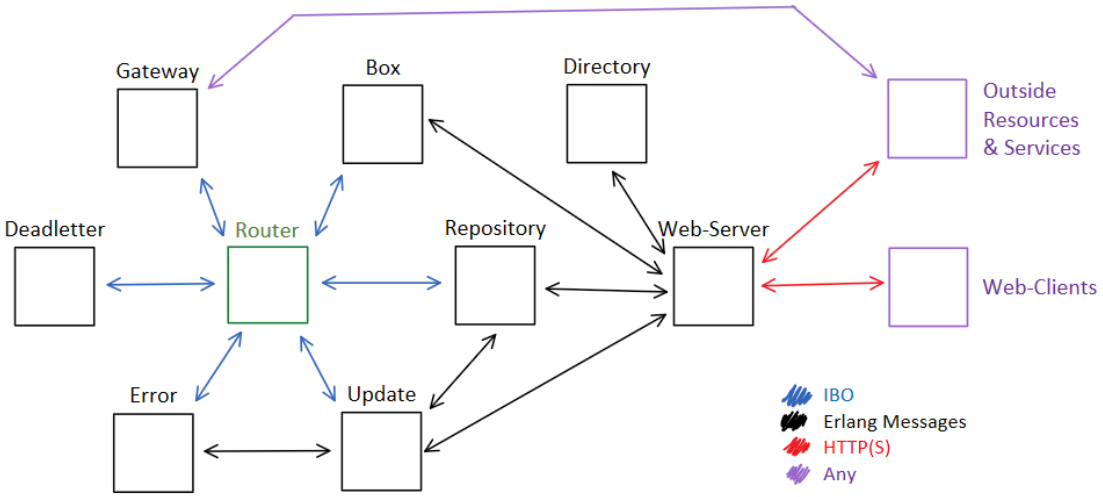

Figure 49 - Resulting logical architecture

Each actor has its own defined API, which can be accessed by other actors. The API is written in a way, that the least amount of data is transmitted between the actors and the interdependence between the actors is kept to a minimum. For instance, the directory actor contains all user and group information and the box actor stores IBOs depending on the group. But the box actor does not need to access the directory actor, but simply provides IBOs for the requested groups, as the actor trusts the requesting actor to know what it is doing.

3.15.2 Physical Architecture

The architecture that is perceived by the actors is quite different to the actual physical architecture. This is due to the fact, that the actors can be distributed within a network of several Erlang nodes. Connected erlang nodes form a fully connected network, which enables all nodes to communicate directly, but it also limits[25] the amount of Erlang nodes that can be used in a cluster of nodes. As the connection can fail between the nodes, each node needs to have its own watchdog and deadletter actor. Each node can either run on a separate computer, but it is also possible to run several nodes on just one computer as well. [13]

[25] By using gateway actors, it is possible to connect several clusters together, effectively circumventing the limit. Further information can be found at [13].

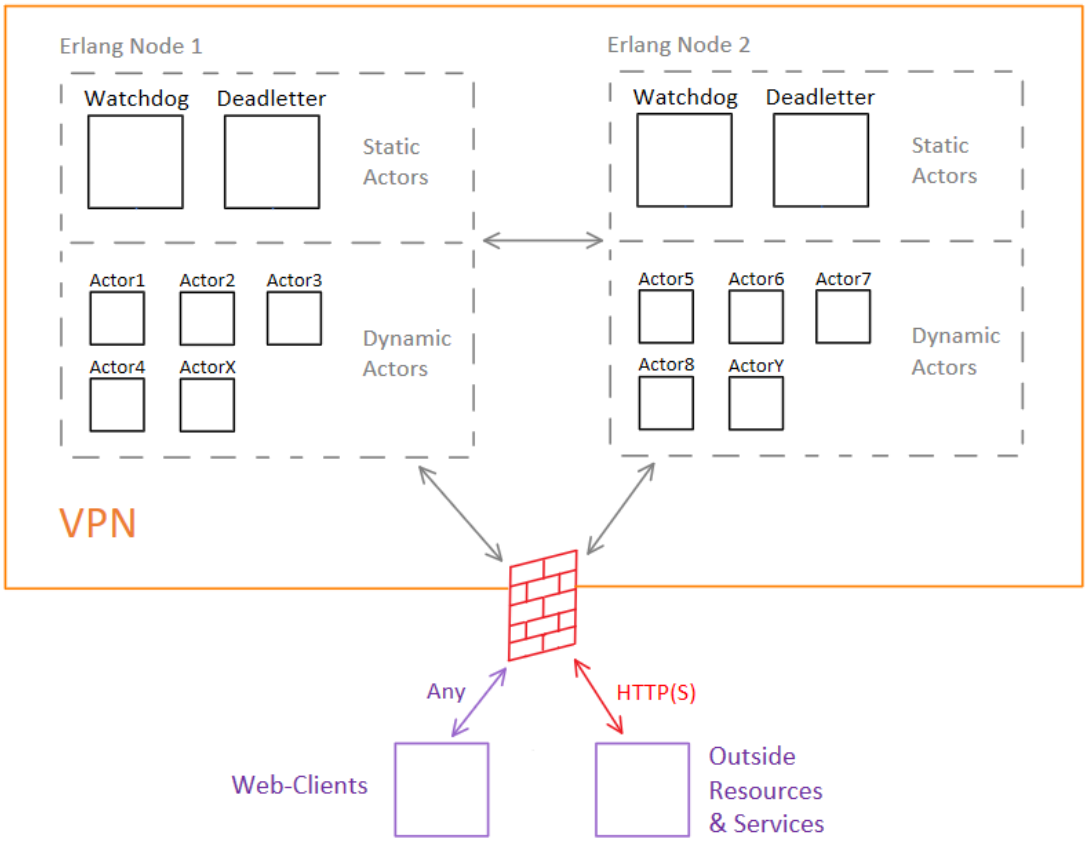

Figure 50 - Resulting physical architecture

Figure 50 is an example, how the actual system can or should be configured in a network. The system itself should be run in a secured Virtual Private Network (VPN) protected with a firewall. Access to the system from outside should only be possible via the web server and gateway actors (also see Figure 49), because Erlang on its own is not secure. This was perfectly illustrated by Fred Hebert in Figure 51, which depicts Erlang's security model.

* This box intentionally left blank *

Figure 51 - Erlang's security model [13]

This concludes the chapter about the system's architecture. The next chapter explains the prototype, which is based on this architecture in order to show the feasibility of the system.

4 Using the prototype

The Erlang prototype does not contain all features portrayed in chapter 3 - IBO System Architecture, but enough to demonstrate the creation and execution of processes.

4.1 Prerequisites

At the time of writing this thesis, no installer has been created that installs all the needed dependencies for the prototype automatically. In order to start and test the prototype, the listed dependencies have to be installed. It is also advisable to install the necessary development tools, which were used to create the prototype.

4.1.1 Git

The source code of the prototype is stored in the following GitHub repository: https://github.com/citrusO2/IBO. The repository was used throughout the development of the prototype and therefore contains the complete prototype.

To download the necessary source files and the dependencies, the version control tool “git” needs to be installed. Usually, programmers have already installed git, as it is a very common tool. Otherwise, a good instruction manual can be found here: https://git-scm.com/book/en/v2/Getting-Started-Installing-Git. The installation is complete, when the command “git” is found in the shell of the operating system.

4.1.2 Installing Erlang

Depending on the used operating system, the installation of Erlang differs. Erlang provides the necessary downloads and instructions on its download page at http://www.erlang.org/downloads.

The prototype has been created with version 18.0, but all versions after 18.0 should be compatible as well. The prototype has also been tested on the latest version (19.0). The installation is complete, when the command “erl” starts an erlang console in the shell of the operating system.

4.1.3 Installing rebar

In order to install the necessary dependencies later, the erlang build tool rebar[26] has to be installed as well. The official version and instructions can be downloaded from https://github.com/rebar/rebar, which is recommended for Linux/Unix/OS X operating systems. For Windows operating systems, the prototype GitHub repository contains a compiled version and a start-script in the folder /tools/rebar, which needs to be copied on the local disc and the path variable needs to be set correctly, so that the operating system can find rebar.

The installation of rebar was successful, when the command “rebar” is found in the shell of the operating system.

[26] There is also a newer version of rebar available (rebar3), but some dependencies have not been compatible with the new version at the time of the prototype development.

4.1.4 Installing IntelliJ IDEA

IntelliJ IDEA is a development environment, which simplifies the development of software. It is also one of the few development environments that supports Erlang. The community edition can be downloaded for free at https://www.jetbrains.com/idea/ and a manual, how to set up the IntelliJ can be found at https://www.jetbrains.com/help/idea/2016.2/getting-started-with-erlang.html. But it is not necessary to download and install IntelliJ to use the prototype, but it is beneficial for analysing the code.

4.2 Basic Usage

This chapter explains the basic usage of Erlang and the prototype. It is not necessary to know Erlang to use the prototype, but the basic configuration has to be supplied via Erlang function calls in an Erlang console.

4.2.1 Starting the Prototype

After the source code project is downloaded from GitHub and all the necessary prerequisites are installed, the start-script "erl_startscript.cmd" can be executed (Listing 1).

```
c:\path\to\project\IBO> erl_startscript.cmd
```

Listing 1 - Starting the Erlang console via the startscript

This small script executes a couple of necessary steps, which are required before the start of the prototype. First, it downloads all other necessary libraries from GitHub and stores them in the /deps folder of the project. Then, it compiles the downloaded libraries and the prototype. At the last step, it starts an Erlang console and starts dependent Erlang applications, which need to be running, before the prototype can be started. But the script does not start the prototype itself.

Before starting the prototype for the first time, the database has to be initialised[27]. This is done via calling the install function of the prototype application in the Erlang console (Listing 2). As also seen in Listing 2, Erlang console commands have to be finished with a dot (".").

```
(console@127.0.0.1)1> ibo_app:install().
```

Listing 2 - Installing the database

When the dependencies are running and the database is initialised, the prototype can finally be started. This is done via Erlang's start function from the application module. As it can be seen in Listing 3, the prototype automatically starts the deadletter actor and the watchdog actor, because they are integral parts of each node. It is not possible to start the prototype without these two actors.

[27] The start-script already started the database, but to initialise the necessary settings, the install function needs to stop the database. After the initialisation, the database is started again.

```
(console@127.0.0.1)2> application:start(ibo).
deadletter_server starting
watchdog_server starting
ok
```

Listing 3 - Starting of the prototype

To speed up the use of the prototype, test data can be installed, which add a couple of actors to the system as well as creating test users and test groups (Listing 4). It should be noted that test data should only be installed once, because the install function would try to install the same actors with the same name, which would create a naming clash.

```
(console@127.0.0.1)3> ibo_app:install_testdata().
directory_server (<<"DIRECTORY1">>) starting
error_server (<<"ERROR1">>) starting
box_server (<<"BOX1">>) starting
xbo_router (<<"ROUTER1">>) starting
repo_server (<<"REPO1">>) starting
web_server (<<"WEB1">>) starting
```

Listing 4 - Installing testdata

With the last command the prototype's basic setup is complete. To stop the prototype, the command "q()." can be used, which shuts down Erlang and the prototype gracefully. The next time when starting the prototype, the two install commands have to be skipped, because the settings are loaded from the disk.

4.2.2 Adding and removing actors

Apart from the mandatory actors watchdog and deadletter, all other actors can be started and stopped dynamically. The adding and removing is done via the watchdog actor, which monitors the active actors. The watchdog actor also keeps a list of the current actor configuration, which is used to start the same actors again, after the whole system is shut down.

```
(console@127.0.0.1)1> watchdog_server:start_iactor(directory_sup,
<<"DIRECTORY2">>).
directory_server (<<"DIRECTORY2">>) starting
ok
(console@127.0.0.1)2> watchdog_server:start_iactor(xbo_router_sup, #{ name
=> <<"ROUTER2">>, allowed => [<<"BOX1">>]}).
xbo_router (<<"ROUTER2">>) starting
ok
(console@127.0.0.1)3> watchdog_server:stop_iactor(<<"DIRECTORY2">>).
directory_server (<<"DIRECTORY2">>) stopping
ok
(console@127.0.0.1)4> watchdog_server:stop_iactor(<<"ROUTER2">>).
xbo_router (<<"ROUTER2">>) stopping
ok
```

Listing 5 - Starting and stopping actors

Listing 5 shows the basic adding and removing of actors. Actors are not started directly, but are started via their supervisor, which also restart the actor in case of an error. That is why the first parameter is the actor's supervisor's name, which is directory_sup for directory actors and xbo_router_sup for routers. Starting an actor with the same name, which is already running, will result in an error (Listing 6).

```
(console@127.0.0.1)5> watchdog_server:start_iactor(directory_sup,
<<"DIRECTORY1">>).
{error,"watchdog already started iactor"}
```

Listing 6 - Trying to start an actor with an already in use name

The second parameter is either the name of the new actor, when the actor does not need additional configuration parameters besides the name (Listing 5, line 1), or a map with additional settings, when the actor needs to be configured more extensively (Listing 5, line 2). The name of the actor has to be a unique global name, which means that it has to be unique on all the nodes in the network, as the names are used to identify the individual actors. When an actor is added or removed successfully, they are automatically added or removed from the configuration file (watchdog_configuration in the root directory of the project), which is automatically created by the watchdog actor.

4.2.3 Actor configurations

Dynamic actors can be configured on startup by supplying a map with configuration parameters. The configuration parameters depend on the actor itself, but each dynamic actor needs at least a global name. Table 1 shows the different actors and how they can be configured. Parameters in brackets are optional.

ACTOR	ACTOR IDENTIFIER	PARAMETERS	EXAMPLE CONFIGURATION
BOX	box_sup	name	`#{name=> <<"BOX1">>}`
DIRECTORY	directory_sup	name	`#{name=> <<"DIRECTORY1">>}`
ERROR	error_sup	name	`#{name=> <<"ERROR1">>}`
ROUTER	xbo_router_sup	name allowed	`#{name=> <<"ROUTER1">>, allowed => [<<"BOX1">>]}`
REPO	repo_sup	name router error managegroups	`#{name => <<"REPO">>, router => [<<"ROUTER1">>], error => [<<"ERROR1">>], managegroups => [<<"MyAllowedGroup">>]}`
WEB	web_sup	name directory box repo (port)	`#{name => <<"WEB1">>, directory => <<"DIRECTORY1">>, box => <<"BOX1">>, repo => <<"REPO1">>, port => 8080}`

Table 1 - Actor start configurations

The corresponding parameters are explained in Table 2 below. Using different types than the ones indicated can result in errors, crashes and abnormal side effects.

PARAMETER	TYPE	EXAMPLE	DETAILS
NAME	binary string	<<”BOX1”>>	All names are given as a binary string, as there is no further conversion needed for the webserver
ALLOWED	list of binary strings	[<<”BOX1”>>, <<”BOX2”>>]	The IBOs are only forwarded by the router to these xActors (Error actor is exempt, as it is used for debugging, while the repo actor is not affected, as it does not receive IBOs)
ROUTER	list of binary strings	[<<”ROUTER1”>>, <<”ROUTER2”>>]	List of routers that will be used for the generated IBOs
ERROR	list of binary strings	[<<”ERROR1”>>, <<”ERROR2”>>]	List of routers that will be used for the generated IBOs
MANAGE-GROUPS	list of binary strings	[<<”MyAllowed Group”>>]	List of groups, which are allowed to store and modify templates (the setting of who is allowed to start a template is stored per template)
DIRECTORY, BOX, **REPO**	binary string	<<”BOX1”>>	The fields directory, box and repo are the global names of the corresponding actors. The web actor accesses these actors by the given names
PORT	integer	8080	The port that is used, has to be unique for each actor and computer (= port must not be in use)

Table 2 - Actor start configuration parameters

4.2.4 User and group configuration

Actors rely on group and user information provided by the directory actor. New users and groups can be created and updated either through a configuration function, like shown in the file src/_example_config/directory_exampleusers.erl, where the whole configuration can be given in a function, that can later be called in the console (Listing 7).

```
(console@127.0.0.1)1> directory_exampleusers:install(<<"DIRECTORY1">>).
ok
```

Listing 7 - Installing users and groups via an install function

As an alternative, users and groups can also be created and updated via calling write_group and create_user or update_user[28] (Table 3 and Listing 8). Write group will overwrite an existing group with the same name, while create_user will check, if a user with the same username already exists and only allow it to be overwritten, when the supplied password matches the stored password[29].

FUNCTION	PARAMETER	TYPE	DETAILS
CREATE_USER, **UPDATE_USER**	Directory Username Firstname Lastname Groups Password	binary string binary string binary string binary string list of binary strings binary string	The directory is the global name of the directory, where the user will be created/updated. The username is used as the user's ID. Groups is a list of group names, which the user is a member of. The server will use the groups for authorisation and will also retrieve inherited groupmemberships.
WRITE_GROUP	Directory Name Description Parents Is_rulegroup	binary string binary string binary string list of binary strings true/false	Like in the functions above, the directory is the name global name of the directory, where the command will be executed. Name is the unique groupname. Parents is a list of groupnames, that are later resolved for the users' inherited groupmemberships. The last parameter is a flag, that is used to create rulegroups, which should be used for special permissions like managegroups for the repo server.

Table 3 - Parameters for creating and updating users and groups

[28] The functions create_user and update_user work exactly the same, they are just a syntactic sugar, so that the code is easier to read.

[29] The password is stored as a hash, so the supplied password will also be hashed before the comparison.

```
(console@127.0.0.1)2> directory_server:create_user(<<"DIRECTORY1">>,
<<"rs">>, <<"Robo">>, <<"Sibing">>, [<<"acme">>,<<"mar
keting">>], <<"MySuperSecretPassword">>).
ok
(console@127.0.0.1)3> directory_server:write_group(<<"DIRECTORY1">>,
<<"sales">>,<<"Sales Department Group">>,[<<"acme">>]).
ok
```

Listing 8 - Creating users via console

The directory actor and other actors have more functionalities than described here. They can be used and are described in detail in the source code.

4.2.5 Web client

With all the actors in place and configured, the system can finally be accessed via a web browser. If the port is not defined in the web actor, the standard port 8080 is used. If the prototype was started locally, the web server can be accessed via the address http://127.0.0.1:8080, as shown in Figure 52. The web client has a responsive layout, so that it fits on most screens[30]. When accessing the website, the web client will automatically try to retrieve user information, using a previously stored session cookie. If the login via the session cookie is not successful, the web client redirects the user to the login window.

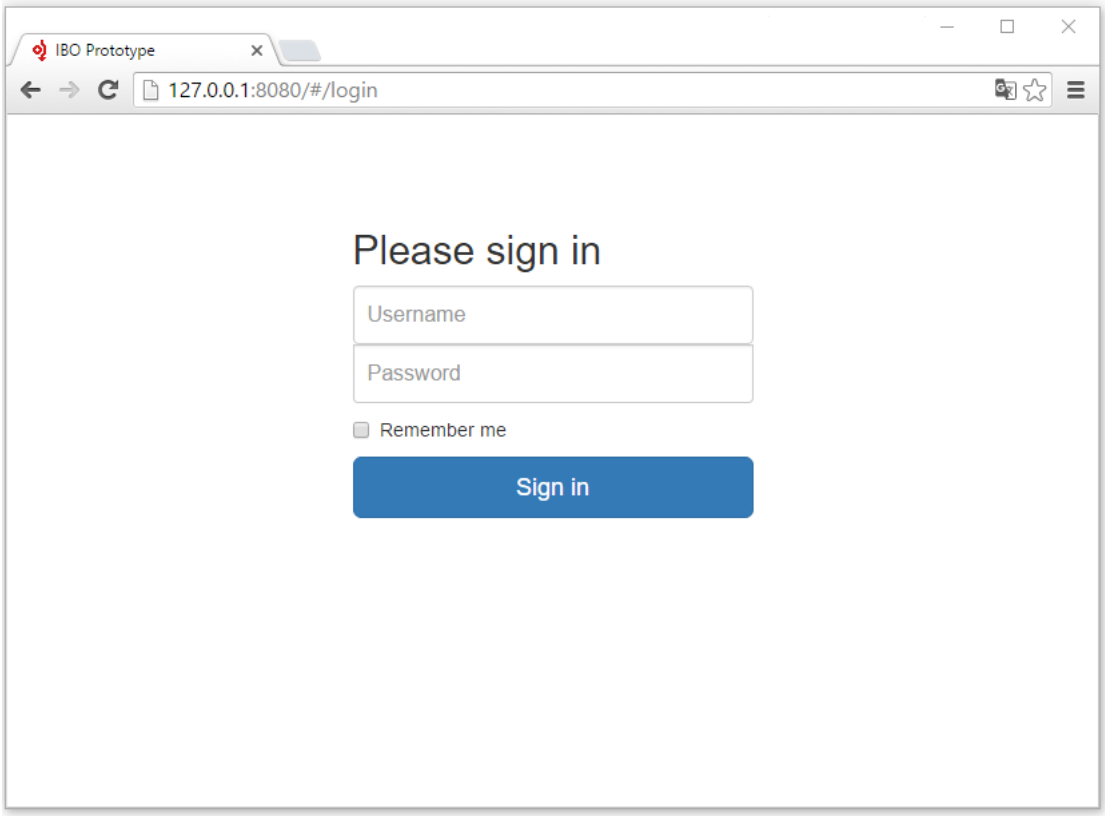

Figure 52 - Login window

[30] Creating new process templates is not optimised for smaller screens.

4.2.5.1 Login

When logging in with username and password, the web actor will try to authenticate against the directory actor, that was configured in the web actor. When the authentication is successful, the web actor stores the user information[31] from the directory actor in two temporary tables, including a session id. The session id is then given to the web client as a cookie, which is used for further authentication.

Using a session id instead of username and password for each request reduces the interdependence between directory actor and web actor and also reduces the computation time, as the password does not need to be rehashed for every single request. This is important, because strong password hashes are deliberately made to be computationally expensive.

4.2.5.2 Logout

To log out, all it takes is to click the logout button, placed on the right side in the top menu bar. But apart from the manual logout, users can also be logged out for other reasons. In order to prevent the same user account to be logged in on different computers, only one session can be stored per user. As soon as the user authenticates via username and password, a possible existing session is overwritten, which invalidates the previous session and forces the logout on the previously used computer. Another reason for getting logged out is, when the Erlang node gets restarted, as the session information is only stored in temporary in-memory tables.

4.2.5.3 Overview

After login, the user lands on the overview page, which lists all available tasks for the currently logged in user (Figure 53). They are ordered per group stored on the box actor[32]. In the example below, the user Robo Sibling is logged in and is presented with the task “Opinion”. The same task is also shown in the left sidebar, which is used for the navigation.

[31] Before the web actor gets the user data from the directory actor, the directory actor removes the hashed password from the user data.

[32] The box actor does not communicate with the directory actor, so the possible groups are created on the box server by the given groupnames in the IBO, which is determined by the field local (also see chapter 2.3.12 - Step - Local). That implies, that the groups shown in the overview can be fewer than the user is a member of.

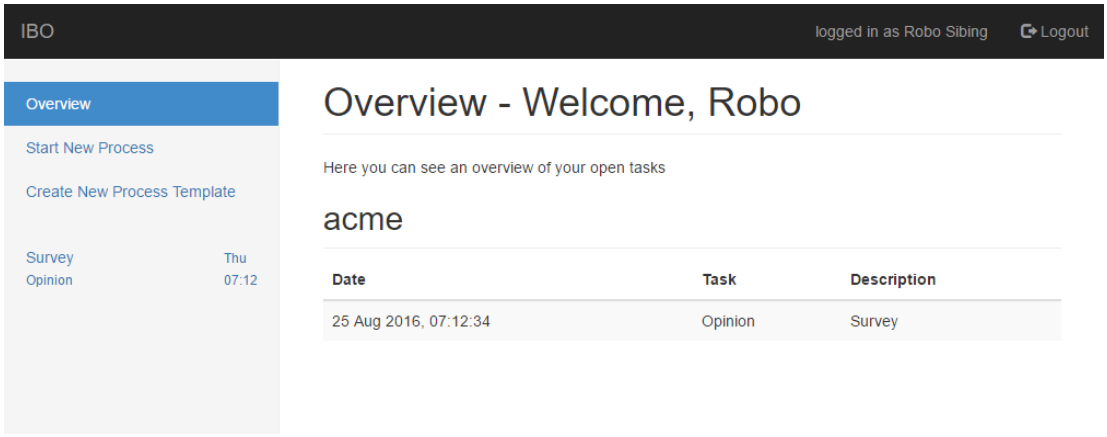

Figure 53 - Overview window

The user can only see open tasks and no completed tasks. All tasks are ordered by group and every member of the group can see the same task. At the moment, there is no task allocation implemented[33], which means that the first user who completes the task "wins". This should be changed in a future version of the prototype, as it can lead to multiple users trying to complete the same step, wasting time. A possible implementation could be, that the input of a user is mirrored to all other users, who also have the same task open. That way all users see all changes, similar to working collaboratively on an online office document.

4.2.5.4 Starting a new process

Users can also start new processes[34], depending on the stored IBO templates. The listed processes (Figure 54) are the processes that the user can start. Here, users can also delete or edit processes, but when a user does not have the necessary permissions, the request is denied.

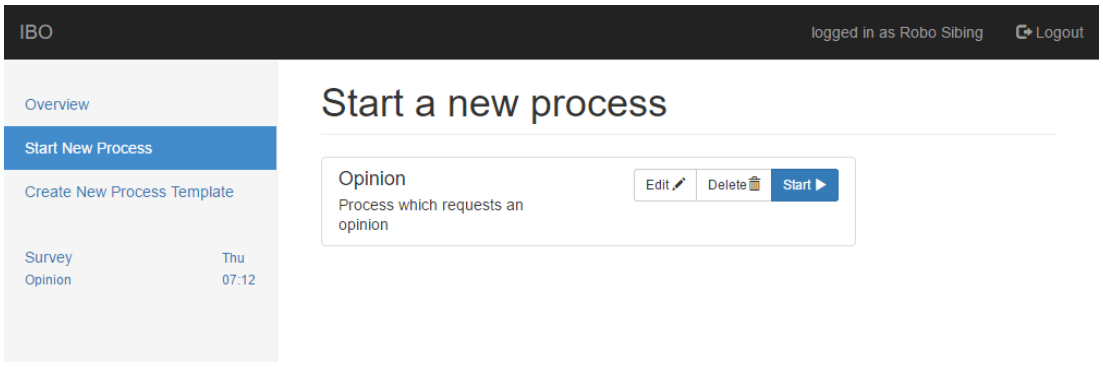

Figure 54 - Start a new process window

[33] For a list of possible resource allocation patterns, see [14].

[34] From a technical point of view, only a new IBO is created and send on its way, but for normal users the term "start a new process" is easier to comprehend.

Future updates to the prototype should include sorting and highlighting mechanisms for the templates, as the current implementation is not well suited for many templates.

4.2.5.5 Completing a task

After selecting a task from the sidebar, a form is created from the data[35] in the IBO (Figure 55). The example used in the figure simply asks for an opinion. The form is divided into two parts: contextual text, that describes what the user has to do and input fields, that request additional information from the user.

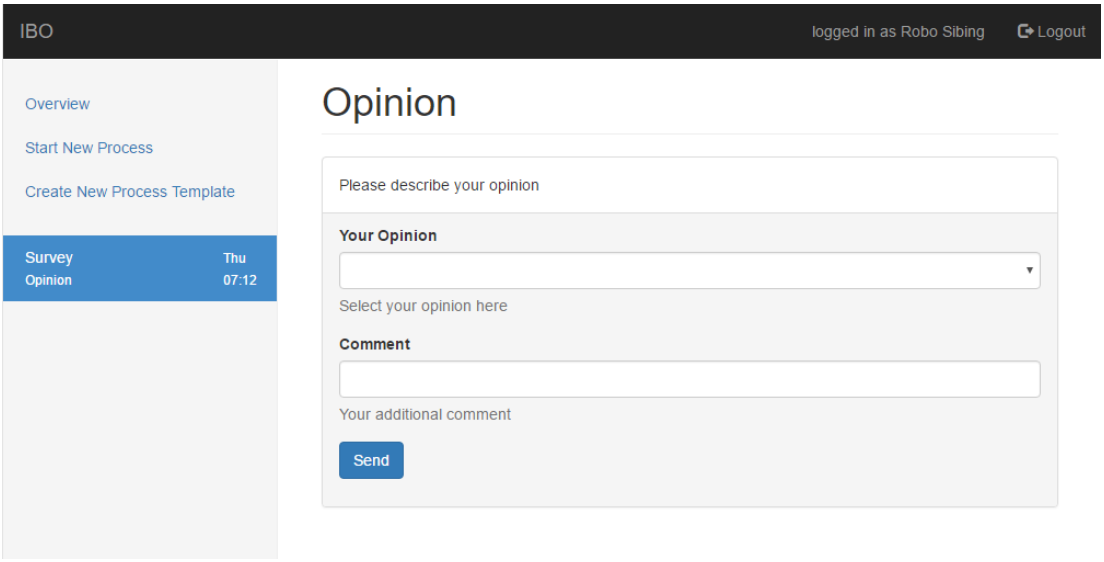

Figure 55 - Task window

The user has to provide the necessary information, which are validated, before they can be send to the server (Figure 56). As web clients cannot be trusted, the data is also validated on the server side, before the step is finally executed on the box actor.

[35] The data structure used to create the form is a JSON schema (http://json-schema.org/). It describes in which form the data has to be in order to be accepted. This can for example contain limitations, how small or big a number can be or if a field is required or not.

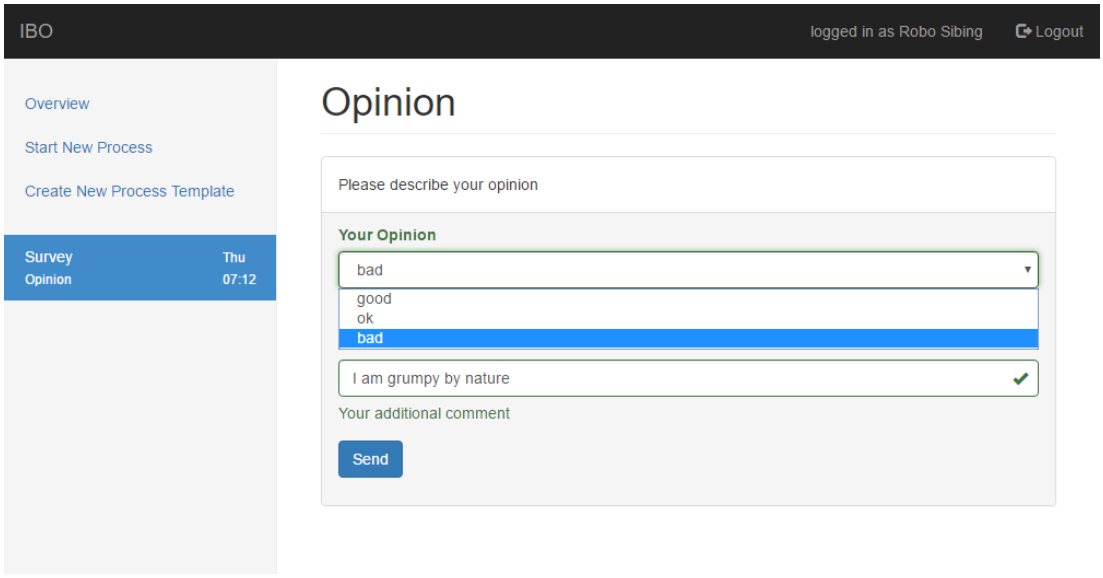

Figure 56 - Task window filled out

The contextual text of the task can also contain dynamic parts, which are created by data provided in previous steps, as can be seen in Figure 57. The figure also shows, that a step can also be purely informative, as no additional data is needed from the user. Additionally, the user Robo Sibling had no access to the next step, which displays the comment from the previous step. That is why a different user is logged in in Figure 57.

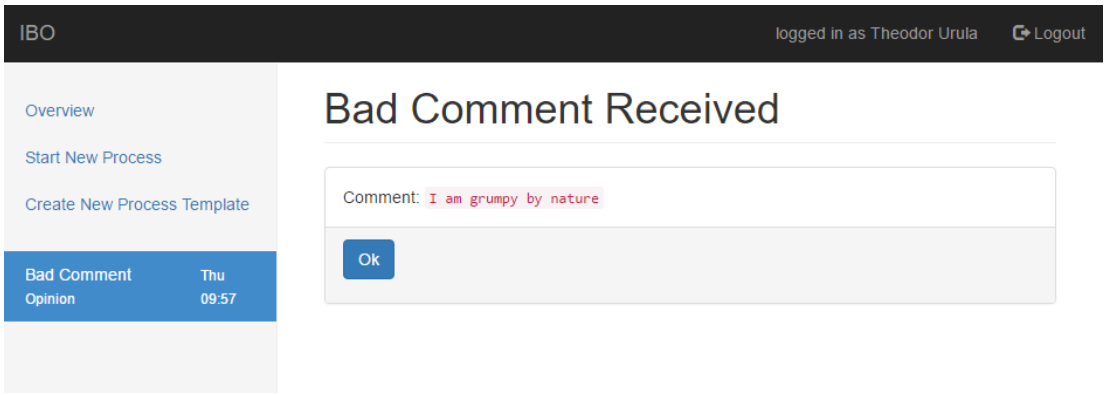

Figure 57 - Task window dynamic text, no additional input

4.2.5.6 Creating a process template

The template creator is the most complex part of the web client. It has to display the structure of an IBO template in a way, that the functionality can be understood easily, but it also has to be as easily modifiable. Having the template created visually further requires transformations between the graphical representation, the corresponding data structure in the web browser and the stored Erlang data structure on the server side, including validations.

The template creator tries to hide the complexity as much as possible, as can be seen Figure 58, where a new empty process template was initialised. The upper area is a

list of xActors[36], which are currently available in the network of nodes (in this case only BOX1 is available). Each xActor is listed with its type (as seen in chapter 4.2.3 - Actor configurations) and its supported libraries[37].

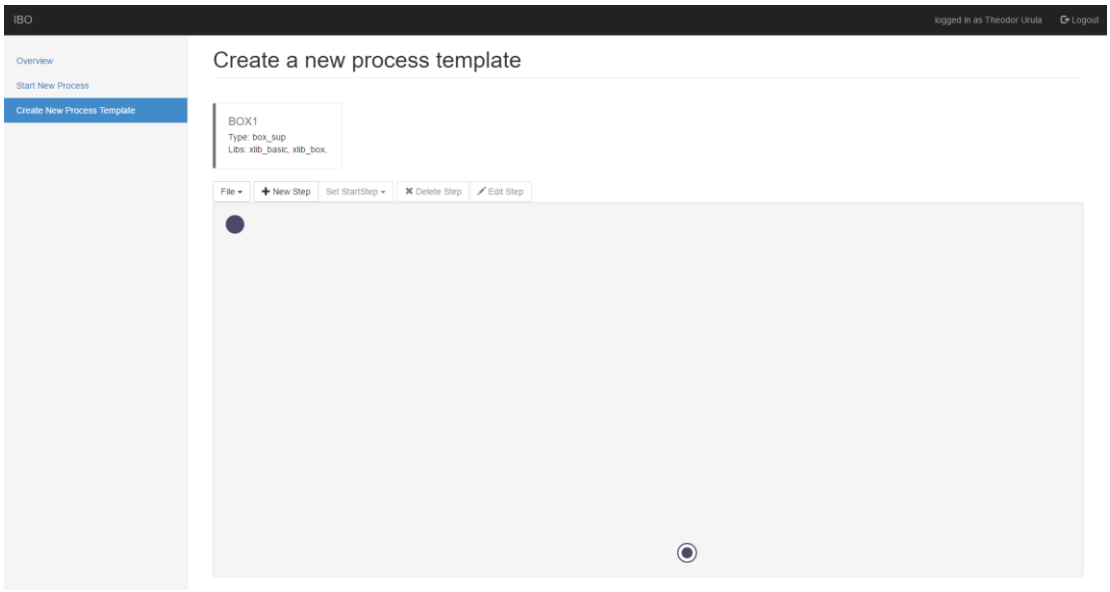

Figure 58 - Template creator window

The template creator uses several menus, which contain and abstract functionalities and which are described below.

4.2.5.6.1 Creating a new step

To create a new step, the "+ New Step" button has to be clicked. A new menu will open, which contains input fields for the description of the step, the domain and the local, referring to the same fields of the IBO structure (see chapter 2.3.9 - IBO - Steps). The domain field defines, which xActor should execute the new step and thus also defines, which commands can be used in this step. However, the user does not know, on which node the xActor is currently running, because it is irrelevant for the definition and execution of an IBO. The value of the local field depends on each actor. For example, the box actor uses the local field in order to define, which user group should have access to the step on the box actor.

[36] The list is compiled by requesting a list of xActors from each connected Erlang node, supplied by the watchdog actors on each node.

[37] Each xActor needs to implement the function xlib_info(), which returns the necessary information for the initialisation of the actor, the supported libraries and all functions in each library. The xlib_info() data is used to configure the GUI for the actor, which simplifies the implementation of new actors into the template creator.

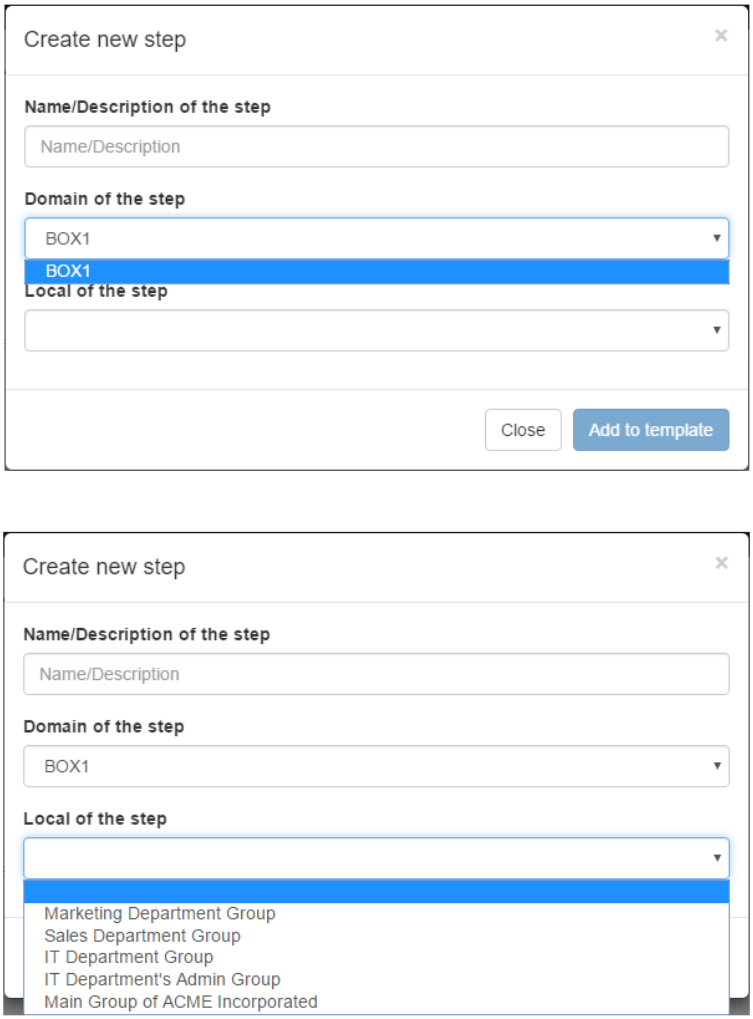

Figure 59 - Create new step windows

After defining all fields, a new empty step is created and displayed in the graphical field, where also the start and finish symbols are. Figure 60 depicts a new empty step called "Request Computer Upgrade", which is executed on "BOX1" and that is accessible for the "it" group.

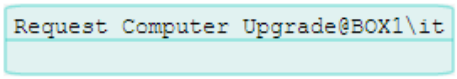

Figure 60 - New empty step graphical representation example

4.2.5.6.2 Editing a step

To edit a step, the step has to either be double clicked or selected by clicking on the step and then clicked on the "Edit Step" button. This will open an editor to change the settings and commands of the step (Figure 61). Everything in the step can be

changed in this window, except the xActor, as each actor can have its own libraries and functions[38].

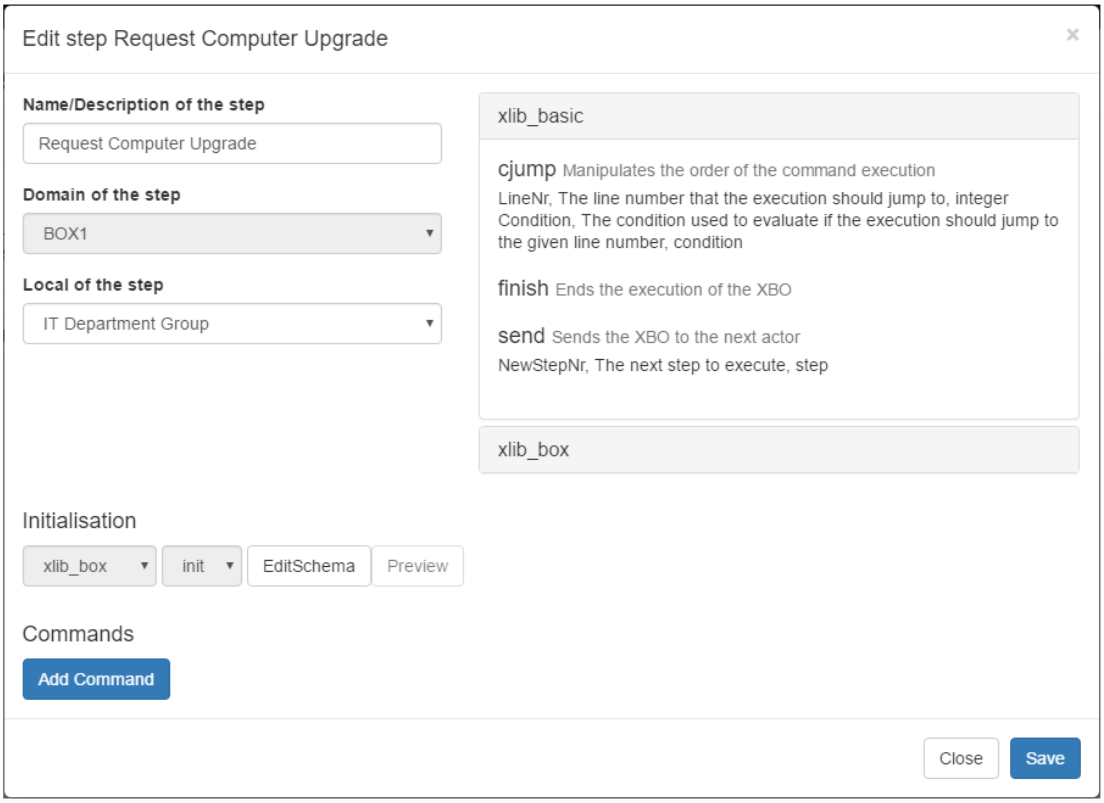

Figure 61 - Edit step window

The edit step window can be divided into three areas: The top left area of the edit window contains the same settings as the create new step window. The top right area displays information about the available libraries, in this case xlib_basic and xlib_box. Each library contains the function, a description of what each function does and its parameters. The third area is the area for the commands, which can consist of a field for an initialisation[39] command and the list of commands which are executed on the xActor.

Clicking on the "EditSchema" button will reveal the schema editor, which is used to configure the user input for this step (Figure 62). The configuration in the schema creates variables, which can be used by the commands of this and other steps. To create a new variable, elements on the right side need to be dragged to the variables section. The order of the variables in the schema will remain the same when the form is filled out later and can be reordered by drag and drop.

[38] Allowing the change of the xActor would make it necessary to delete all commands already set in the step, which could lead to users deleting the commands by accident. Therefore, users have to deliberately create a new step to use a different xActor.

[39] xActors can have an initialisation command, that has to be used. The box actor requires its own initialisation, but not all xActors do need to have one.

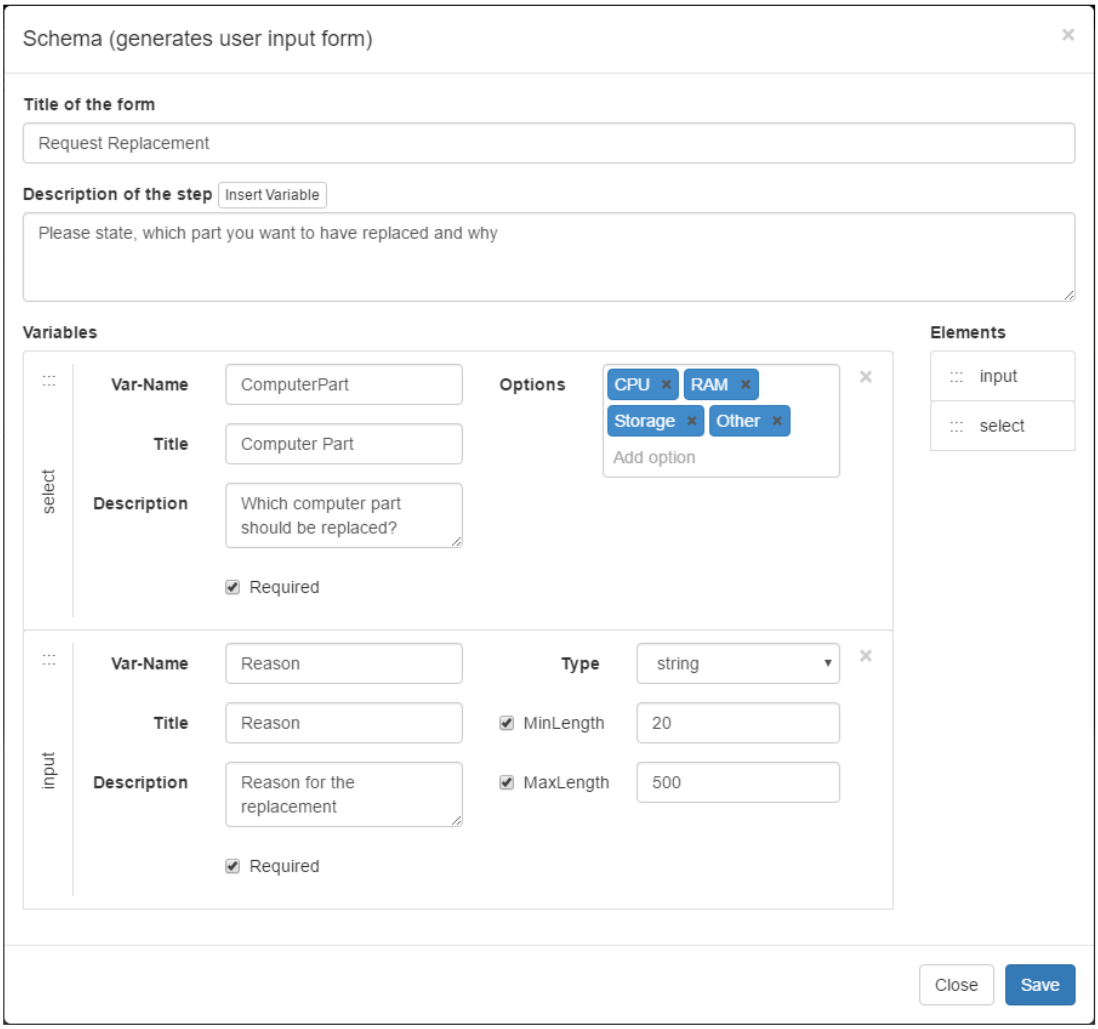

Figure 62 - Schema configuration window

To use variables in the description of the step, the “Insert Variable” button in the schema window can be used (Figure 63). This can be used to display the defined variables from any step in any following step. Using the example of Figure 62, a person tasked to approve or deny the request to replace a computer part, needs to know, which part should be replaced and why. By inserting variable templates into the step description, the box server will replace the variable template and display the data defined in a previous step instead[40].

[40] Figure 57 shows how such a variable template looks for a user completing a task.

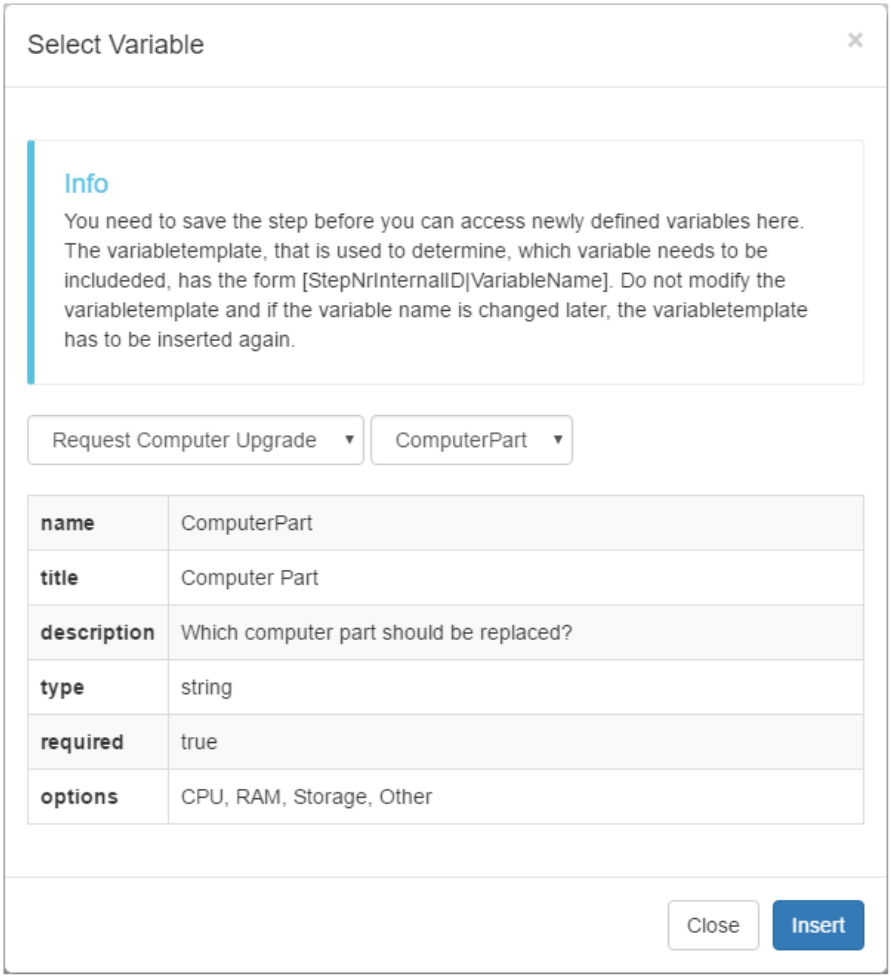

Figure 63 - Select variable window to insert a variable template into the step's description

Editing commands is also implemented in a very straightforward manner. Each command consists of a select box for the library, a select box for the corresponding function and possible additional fields, depending on the selected function. Commands are executed from top to bottom and end, when the command “send” or the command “finish” from the “xlib_basic” library is reached. Thus, every step must contain at least one of the aforementioned functions.

As shown in Figure 64, the simplest possible command is “finish” from the library “xlib_basic”, which ends the execution of the current IBO. However, having only one step with just the finish command is nonsensical.

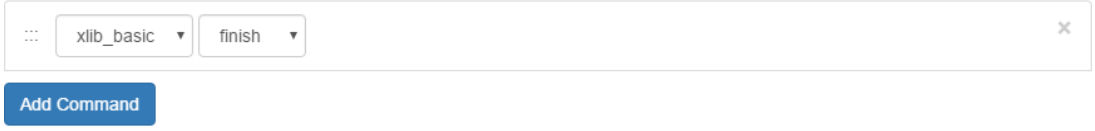

Figure 64 - Commands section of the edit step window - finish execution

Instead of ending the execution after one step, the first step should link to a second step. Thus, a new step has to be created in order to link to it. After creating a second

step, the first step can then send the IBO to the second step, using the command "send" from the library "xlib_basic" [41] as depicted in Figure 65.

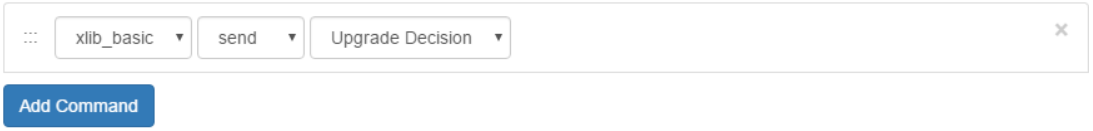

Figure 65 - Commands section of the edit step window - next step

To create non-linear processes, the "send" and the "end" command have to be combined with the command "cjump", which stands for conditional jump. Figure 66 is an example with additional steps[42], which shows, how the command "cjump" can be used to create conditional paths. When the variable "Decision" from the step "Upgrade Decision" is equal to "Approve", then the execution jumps to the third line and sends the IBO to the step "Upgrade Computer". When the condition is not met, the next line "finish" will be executed, which ends the process.

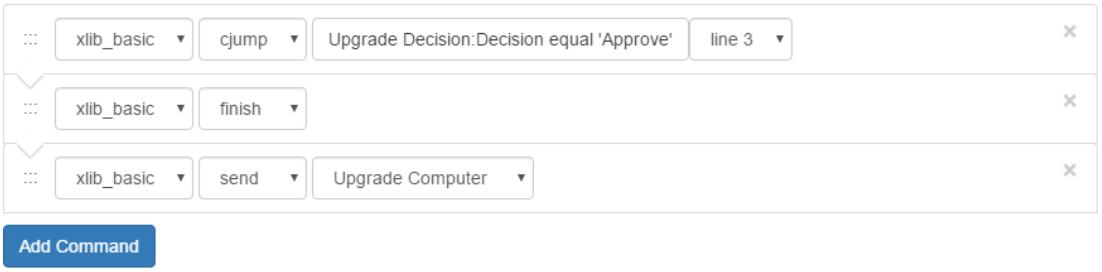

Figure 66 - Commands section of the edit step window - conditional execution

4.2.5.6.3 Setting a start step

Bevor the template can be saved, it needs to be defined, which step should be the first step. This is done by using the "Set StartStep" button from the menu bar. The step, which should be executed first, is stored in the template, but will not be stored in the IBOs generated by the repository actor. The selected step is then indicated by an arrow from the starting point to the first step.

[41] It is also possible to send the IBO to the same step again, but the purpose of that is questionable. When the send command is the only command of the step and the send command points to the same step, an infinite loop is created. Currently, the only way to stop such an infinitely looping IBO is to remove the IBO from the box actor's database.

[42] The complete process is depicted in Figure 67.

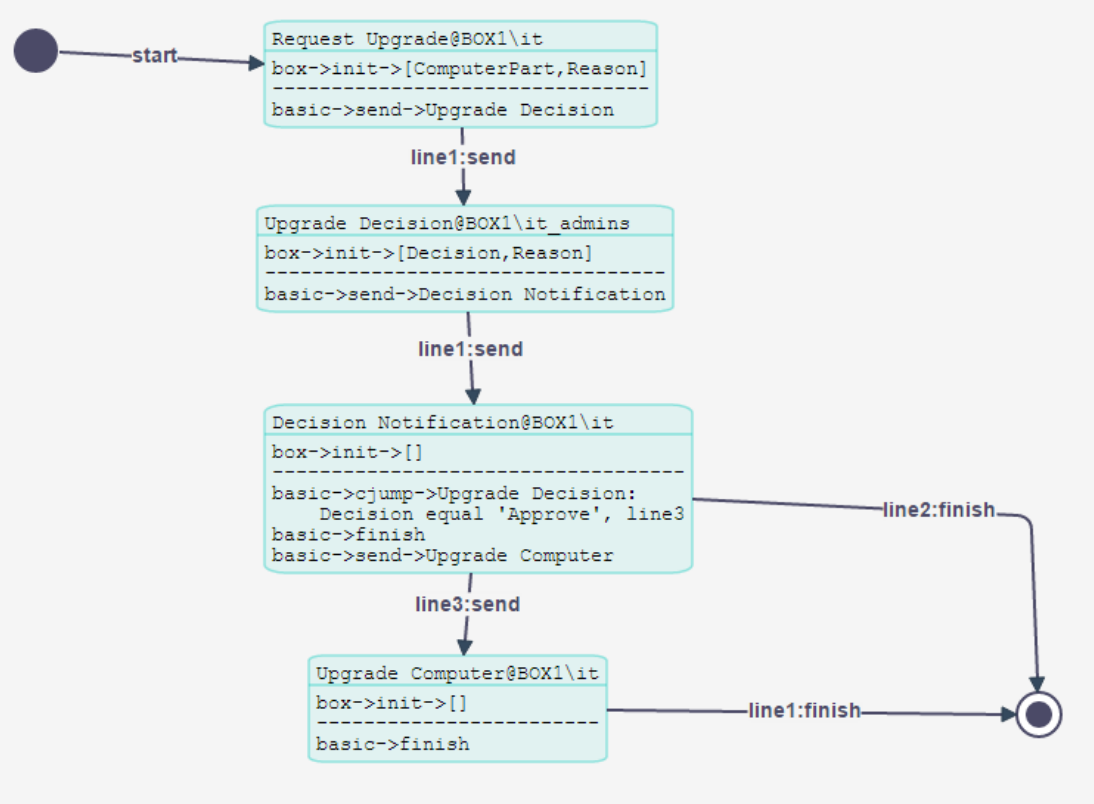

Figure 67 - Example template

All elements can be moved around on the working area, but they cannot be resized manually. Furthermore, it is also possible to change the path of an arrow by simply dragging the arrow into the wanted position. When the template is saved on the repository actor, the positional data is also stored. Hence, the position of each element will be restored when editing the template at a later timer.

4.2.5.6.4 Saving a template

After a template is created, it can be saved by using the “save template” option from the “File” menu item. This will open a new window for the template settings (Figure 68). It contains settings for the template name, a description, the time to live of the later generated IBOs[43] and the groups, which are allowed to start the template.

[43] The time to live will be set when the template is generated, but is currently not used by the actors.

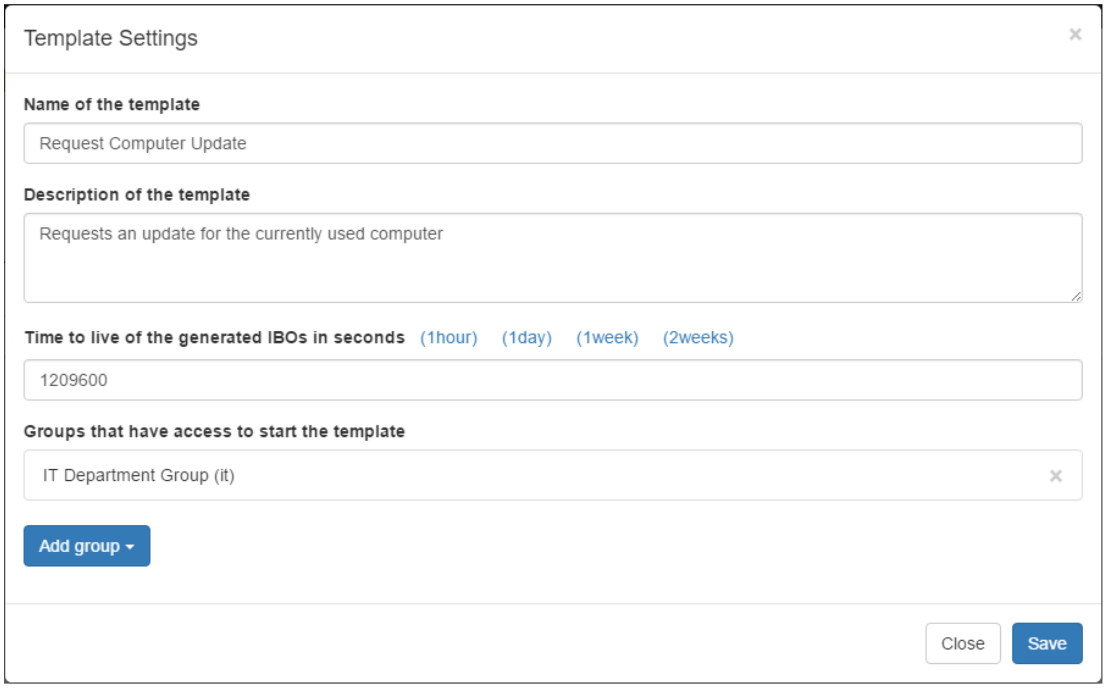

Figure 68 - Template Settings window

4.2.5.7 Updating a process template

A process template can be updated by clicking on "Edit" instead of "Start" when selecting a process (see chapter 4.2.5.4 - Starting a new process). If the user has the necessary access rights, the template is loaded from the repository actor and the graphical representation is initialised. If the necessary actors are not available, the template cannot be edited, because the graphical user interface relies on library information of the used actors. But when the template is loaded successfully, the template can be edited like described in the previous chapter.

4.2.5.8 Deleting a process template

Deleting a process template is like updating a process template, only that instead of "Edit", the button "Delete" has to be used when selecting a process. With the necessary permissions, the template is removed from the list of templates and stored in a separate table, that can be used to restore the template or older versions of the template[44].

4.3 Example Process

To demonstrate, how a process can be modelled with the prototype, an example process is implemented, which allows user to send bug reports. The example processes will also reveal the current limitations and shortcomings and show where and how the prototype should be improved.

[44] Currently there is no user friendly way implemented to restore a template or to roll back a template to a previous version.

4.3.1 Process description

Users send in bug reports, which help the IT department to improve their software. The bug report contains a detailed description of the bug. Then, someone from the IT department reviews the bug report and decides between five different options:

- The submitted bug is indeed a bug and needs to be fixed
- The submitted bug is actually a feature and is nothing that needs fixing
- The submitted bug is a normal behaviour
- The submitted bug cannot be reproduced
- The submitted bus is already fixed

Depending on the judgement from the reviewer, the next step is either to fix the bug or to review the feature. If the bug is already fixed, cannot be reproduced or is a normal behaviour, the process ends. After fixing the bug or reviewing the feature the process ends as well.

4.3.2 Resulting template

Figure 69 shows the resulting template. Each box represents a step in the form of an input field for a particular user.

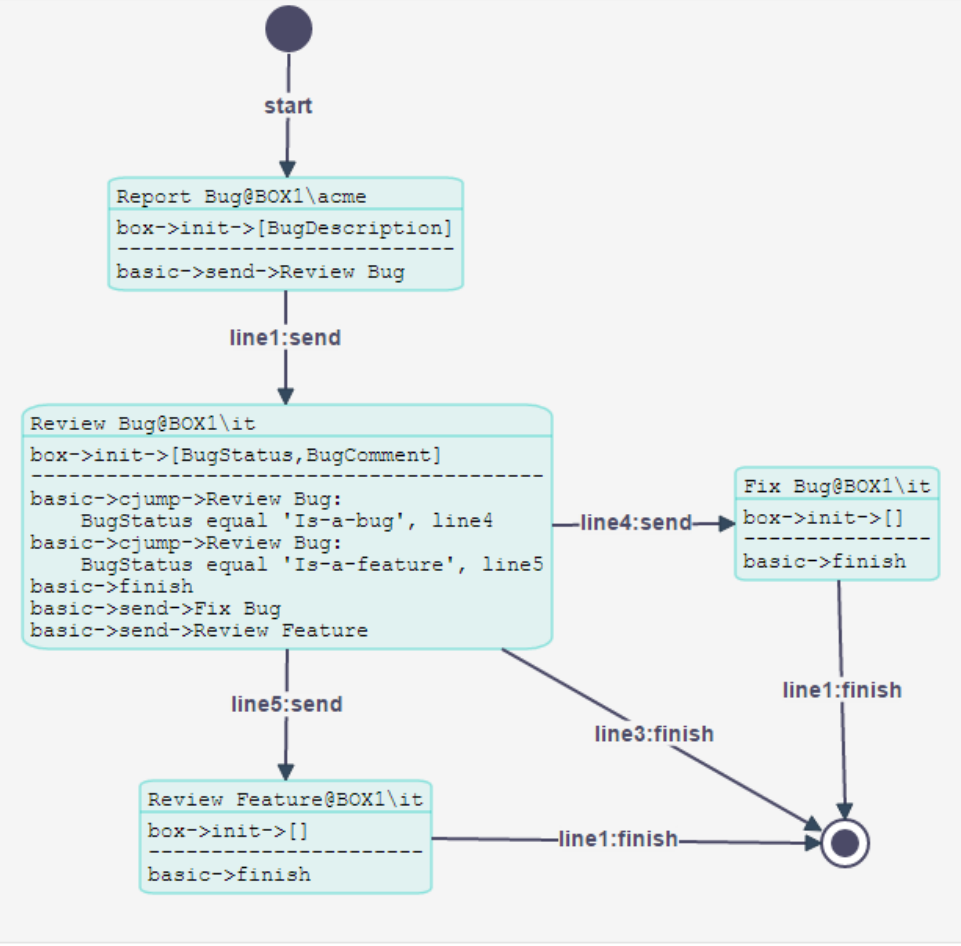

Figure 69 - Bug report example template

4.3.3 Example bug report

The following exemplary bug report shows, how the process looks like for the users. In the example, a user reports, that ALT+F4 closes the application (Figure 70).

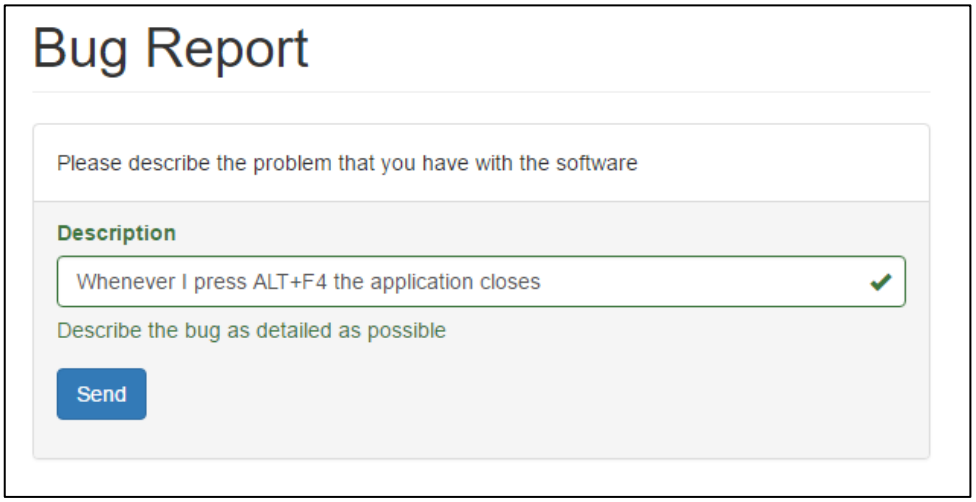

Figure 70 - Exemplary bug report

After sending this bug report, a user from the IT department then reads the bug and evaluates it to be a normal behaviour of the software and not a bug (Figure 71).

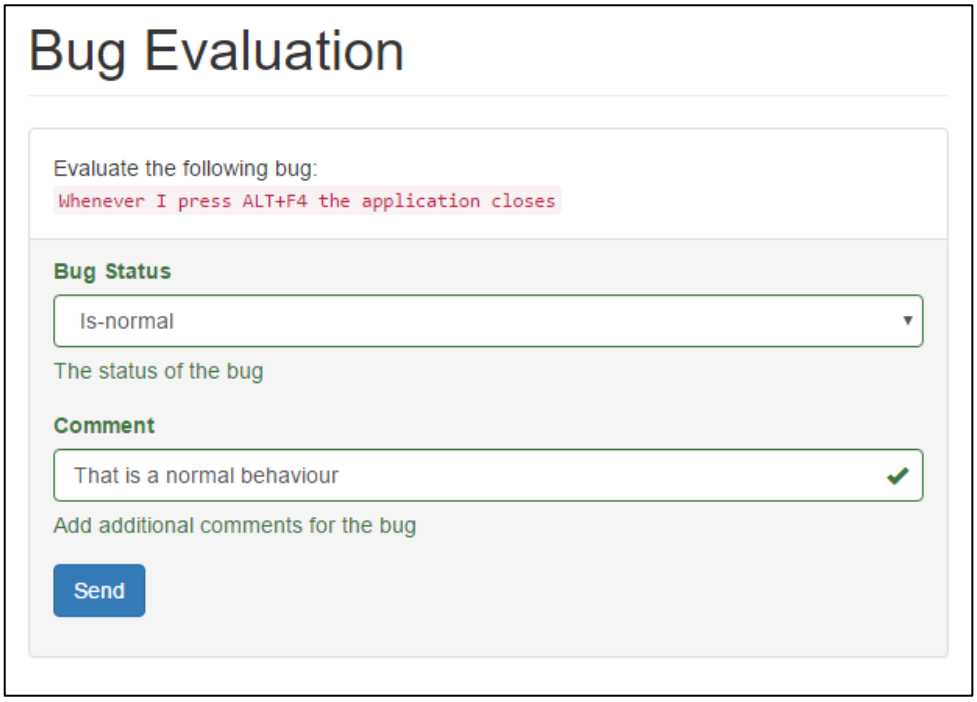

Figure 71 - Exemplary bug evaluation

This then ends the process, as the process execution will run into the “finish” command in the step Bug Review from Figure 69.

4.3.4 Limitations

The use of example processes reveals several limitations that the current prototype implementation has. They start from minor nuisances like not being able to move several boxes at once to the issues described below.

4.3.4.1 Addressing users more directly

The current prototype implementation does not address users directly, but uses groups or roles instead. This should always be the norm for processes, but sometimes it is necessary to directly address a person. When the bug review comes to the conclusion, that the bug cannot be reproduced, then it should be possible to request more information from the person who reported the bug.

4.3.4.2 Timeouts

Currently, there are no timeouts implemented for each step. But especially when addressing individual users instead of a group, it is much more likely that a step can be left unanswered for a longer amount of time. So before users get addressed directly, timeouts[45] should be implemented for each step.

4.3.4.3 Starting of a process

When a user wants to report a bug, the process has to be started before the user can supply data for the bug. But when the process is already started and the user changed his mind and does not fill out the bug report, the process is not advancing. This can be solved by using timeouts, but there is a better way. At the start of the process the user should already provide the necessary data[46] instead of at the first step. This will prevent users from starting processes unnecessarily.

4.3.4.4 Limited amount of commands and actors

Currently, there are only three basic commands and the command to initialise the box actor. To really utilize the system, more actors, libraries and commands need to be implemented. For instance, in the template in Figure 69 the step “Review Bug” could have been created with only three commands instead of one if there was a conditional send command.

Another example of the limited functionality is, that it is not possible to attach files to the bug report like images, which would help describing the reported bug. Furthermore, the router implementation is very limited as well, only forwarding IBOs to their right destination but not actual logging the data, hence there is no tracking of the IBOs available.

4.3.4.5 GUI improvements

Using the graphical user interface also uncovered further improvement potential, which mostly do not affect other parts of the system. Such features include copying

[45] Implementing timeouts should be done when implementing exceptional case handling, as a timeout can also be seen as an exceptional case (referring to chapter 3.13 - Exceptional case handling).

[46] Similar to the box actor, the repository actor should handle user data with a schema before creating a new IBO.

of steps, selecting and moving multiple steps and symbols at once and being able to collapse steps to hide the step's commands. Other features require additional changes to other actors, like being able to rename the directed arrows between the steps and changing the size of the working area, as additional data needs to be stored on the repository actor for that.

4.3.4.6 Parallel tasks

The prototype can handle sequential tasks and can branch into mutually exclusive paths, but it is not possible to execute two tasks at the same time in one single process. Implementing a basic parallel execution is trivial, but there are many side effects that need to be considered. By just copying the IBO[47] and sending it to several actors instead of executing one step after another, the execution becomes parallel. But the real question is how and when the copies are merged together again.

Following the actor philosophy, a possible solution would be to use a splitting actor and a merging actor, which take care of the splitting and merging procedure and its side effects. Designing these actors is a highly delicate challenge, because for instance when the same IBO is copied over and over again, the whole system could get overloaded.

[47] Copying the IBO also has side effects on the routers which log the step data, because step data are not to be overwritten. Thus, having the same IBO multiple times in the system would create errors, which is why the IBO cannot be copied only but has to be modified.

5 Comparison to BPMN

When trying to compare the IBO system with BPMN it is helpful to start with the similarities. The similarity that stands out most is that both have a graph like structure. This is particularly apparent in Figure 72, where an example process is depicted in BPMN and IBO. As it can be seen in the comparison, the basic tasks[48] of the BPMN model are nearly identical to the steps of the IBO template. The main visual differences are the routing and the task assignment: while the send command of IBO links directly between steps, BPMN requires additional gateways in case of conditional routing and IBO assigns the role on a per task basis instead of depending of the placement in pools.

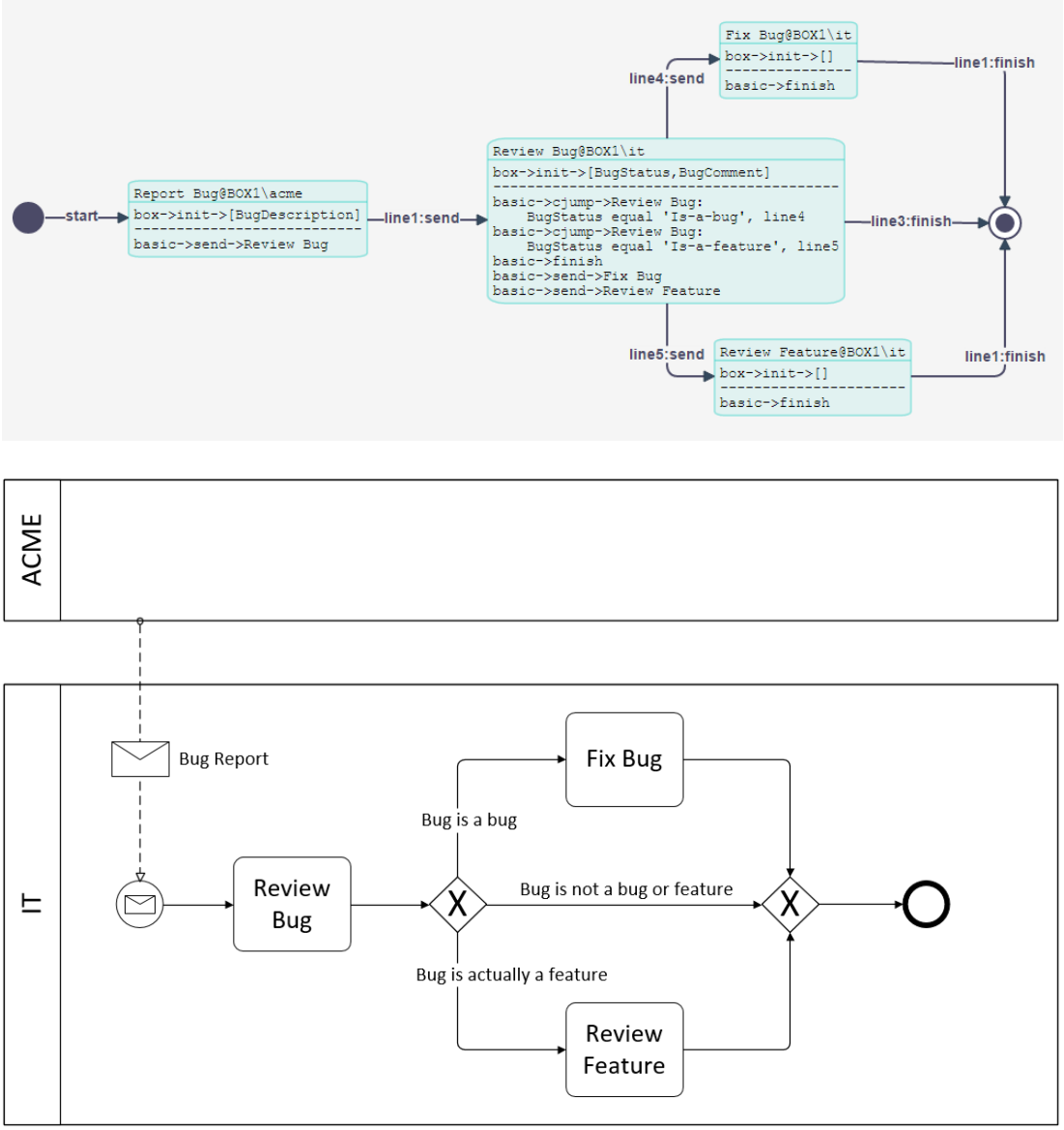

Figure 72 - IBO template vs. BPMN model

[48] In BPMN it is called task, while in IBO it is called a step

Assigning roles to tasks instead of the placement in a pool provides a higher flexibility, but it does not offer the same clear arrangement of pools. But essentially, IBO and BPMN provide the same task assignment functionality with just a different representation. Where they really differ is the send logic, as it is placed after a task in BPMN in contrast to within the step in IBO (Figure 73). Placing that logic in the step is a consequence of optimisation: actors have to send their IBOs to the next actor anyway, so dedicating an additional actor for this task is unnecessary.

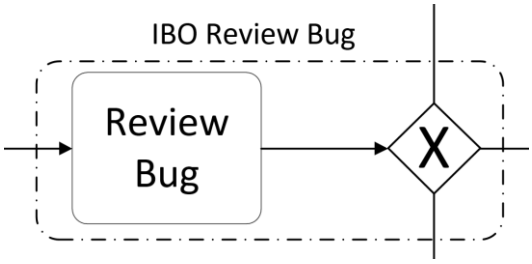

Figure 73 - Scope of IBO Review Bug in contrast to its BPMN task

As the send logic can be placed anywhere within a step in IBO, the whole step does not have to be executed all the time. By placing conditions in the task, it is possible to skip portions of the task and only execute the necessary parts, instead of always executing the whole task. Creating the same behaviour of BPMN with IBO means to simply place the send logic at the end of each step instead of anywhere in between. Hence, IBO provides a higher flexibility at the cost of readability.

The shown differences suggest, that a transformation from BPMN models to IBO templates and vice versa might be possible, though the complexity of the transformation depends on the symbols used in the BPMN model. The used example does not cover complex BPMN symbols, which could be more difficult to implement with IBO. A complete automatic transformation from BPMN models to IBO templates however is not feasible, as individual tasks in BPMN are generally loosely defined[49], unlike the steps of IBO templates. Thus, BPMN models transformed from an IBO template have less precision than the IBO template and IBO templates transformed from a BPMN model are not executable straight away and require additional manual interventions. As a result, a full round-trip[50] from IBO template to BPMN model and back to IBO model seems out of question.

[49] Tasks can have their own sub-processes, which can define a task in more detail, but this does not make BPMN tasks executable.

[50] A full round-trip means, an IBO template generated from a BPMN model is identical to the original IBO template, which was used to generate the BPMN model. Thus, after converting an IBO template to BPMN and back, no information must be lost.

6 Results

The successful implementation of a prototype, which is based on the concepts proposed in this thesis and the creation and execution of processes on the prototype shows, that business processes can be executed on actors via intelligent business objects. The prototype is thus the proof of concept, but this thesis does not show, how well the prototype compares to conventional business process execution. Further research and development is needed in order to compare the concept with other ways of executing business processes, as the functionality of the current prototype implementation is too limited for a significant analysis and assertion.

The future potential of the concept is very high, as the underlying actor model enables a high scalability and flexibility. In addition, the proposed system architecture covers other functionalities, which are generally not possible in conventional business process engines. These include the handling of exceptional cases, ad hoc processes and live updates of business processes while they are executing. Implementing these features is a major task, but could demonstrate the true value of the architecture.

Erlang proofed to be a well suited choice for the prototype, as it enforced the Actor Model and prevented the use of otherwise quick and dirty programming short cuts. Thus, the prototype is already very reliable and easily extensible. Erlang also simplified making the software distributable across several nodes. However, choosing a different programming language would also be possible and would decrease the development time in certain areas, while in other areas the development time would be increased. For instance, when using JavaScript on the server side, no data transformation between the web client and the server is necessary, but it would be much more difficult to distribute the prototype on several computers at once.

The visual structure of BPMN models and IBO templates are similar - it is their focus that differs: BPMN tries to represent business processes in a well-arranged and clear manner, while IBO tries to provide a framework to execute business processes in whatever way needed. At its core, IBO is a dissected process execution engine and the processes are defined with its own high level programming language. But due to the graph oriented nature, it was possible to create a visual representation, which is very similar to BPMN. It might also be possible to use BPMN as a graphical representation of IBO templates, but the BPMN representation could then limit the functionality of the overall system, as IBO templates provide a higher flexibility.

7 Conclusion

Using Intelligent Business Objects to store process information in conjunction with actors proved to be a feasible way of creating a business process engine. The over forty years old actor model was successfully leveraged from a mathematical theory of computation to a basic business process management software. Additionally, the concept could also be applied to other areas like manufacturing. Storing machine configurations and routing data on an unfinished product itself could also help to digitalise manufacturing processes.

Choosing an appropriate technology for the implementation is crucial, as the complexity of the system is increased with every additional functionality. Erlang was a good choice for the development of the prototype, as it enforces the actor model. However, which technology is used is only of secondary importance for users. For them, the complexity of the system is hidden, when they use their browser to design and start processes. How well the new approach fares against existing solutions has yet to be determined, but this work could possibly open up a new chapter in how business processes are executed.

References

[1] M. Rosemann and J. vom Brocke, “The six core elements of business process management,” in *Handbook on Business Process Management 1*, Springer, 2015, pp. 105-122.

[2] M. Weske, Business Process Management: Concepts, Languages, Architectures, 1st ed., Springer Publishing Company, Incorporated, 2010.

[3] M. Chinosi and A. Trombetta, “BPMN: An Introduction to the Standard,” *Comput. Stand. Interfaces,* vol. 34, no. 1, pp. 124-134, #jan# 2012.

[4] J.-J. Dubray, „The Seven Fallacies of Business Process Execution,“ 4 December 2007. [Online]. Available: https://www.infoq.com/articles/seven-fallacies-of-bpm. [Zugriff am 08 08 2016].

[5] M. Rouse, “What is BPEL (Business Process Execution Language)?,” October 2014. [Online]. Available: http://searchsoa.techtarget.com/definition/BPEL. [Accessed 08 08 2016].

[6] B. T. Nguyen, D. H. Nguyen and T. T. Nguyen, “Translation from BPMN to BPEL, Current Techniques and Limitations,” in *Proceedings of the Fifth Symposium on Information and Communication Technology*, New York, NY, USA, 2014.

[7] A. Fleischmann, W. Schmidt, C. Stary, S. Obermeier and E. Brger, Subject-Oriented Business Process Management, Springer Publishing Company, Incorporated, 2014.

[8] W. M. P. van der Aalst, “The Application of Petri Nets to Workflow Management,” *Journal of Circuits, Systems and Computers,* vol. 08, no. 01, pp. 21-66, 1998.

[9] G. Li, V. Muthusamy and H.-A. Jacobsen, “A Distributed Service-oriented Architecture for Business Process Execution,” *ACM Trans. Web,* vol. 4, no. 1, pp. 2:1--2:33, #jan# 2010.

[10] C. Hewitt, P. Bishop and R. Steiger, “A Universal Modular ACTOR Formalism for Artificial Intelligence,” in *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, San Francisco, CA, USA, 1973.

[11] C. Hewitt, “Actor Model of Computation: Scalable Robust Information Systems,” *CoRR,* vol. abs/1008.1459, 2010.

[12] V. Vernon, Reactive Messaging Patterns with the Actor Model: Applications and Integration in Scala and Akka, Addison Wesley, 2015.

[13] F. Hebert, Learn You Some Erlang for Great Good!: A Beginner's Guide, San Francisco, CA, USA: No Starch Press, 2013.

[14] “RFC 791 Internet Protocol - DARPA Inernet Programm, Protocol Specification,” 1981.

[15] N. Russell, A. H. M. Ter Hofstede, D. Edmond und W. M. P. van der Aalst, „Workflow resource patterns,“ 2004.