ulm university universität ulm

University of Ulm | 89069 Ulm | Germany

Faculty of Engineering and Computer Science
Institute of Databases and Information Systems

A Process Engine Independent Architecture Enabling Robust Execution of Mobile Tasks in Business Processes

Master Thesis at the University of Ulm

Submitted by:
Steffen Musiol
steffen.musiol@uni-ulm.de

Reviewers:
Prof. Dr. Manfred Reichert
Prof. Dr. Thomas Bauer

Adviser:
Rüdiger Pryss

2014

Copy April 15, 2014

Composition: PDF-$\LaTeX\,2_\varepsilon$

Abstract

With the fast improvements of mobile technology in the last century, the importance of mobile and pervasive computing has increased in all disciplines of computer science. The new technologies have set the stage for a whole set of new applications (e.g., medical- [8][3] and business-applications). Since Business Process Management Systems (BPMSs) are an established and widely used technique in various businesses and industries to manage recurring workflows and to support people in performing their tasks efficiently, the integration of mobile support into a business process environment is desirable.

In prior research work, a framework to foster the execution of business processes on mobile devices was introduced. It describes a life cycle for mobile tasks based on an automated delegation mechanism and a backup operation for escalation handling.

In this work, the framework will be extended to cope with crucial shortcomings such as the execution of mobile tasks in an unreliable network. Using this, an architectural concept for integrating mobile processes into an existing business process environment will be introduced. A three layer architecture that introducing an intermediate service layer for the mobile task execution will be used to minimize the impact on underlying systems. Additionally, a prototype has been implemented, which will be evaluated against three BPMSs, to prove the feasibility of this approach.

Contents

1 Introduction and Motivation

Mobile and smart devices are becoming increasingly important in today's life. The amount of worldwide smartphone ownerships has been increased by 21% within the last two years [1]. With high investments in the development of new mobile technologies, the devices have become more powerful and are replacing the classic PC in many situations of our daily life by providing flexibility and constant availability of information. This trend does not stop at the private sector. Concepts like “Bring your own device” (BYOD) [2] and the increasing amount of mobile enterprise applications are showing, that business and knowledge workers want the same flexibility for their work as they have in their private life. In other sectors, mobile technologies open up possibilities for streamlining and optimizing existing processes by making information ubiquitously available and put them into the current context. For example, Pryss et al. [3] propose a mobile infrastructure to support physicians and nurses on daily ward rounds.

Business Process Management Systems (BPMSs) are an established and widely used technique to manage recurring workflows and to support people in performing their tasks efficiently. There are applications in business, industrial and public sectors, that have a wide range of different requirements (e.g., an ordering procedure or a medical examination). BPMSs are providing tools to map these workflows on a process model (e.g., BPMN 2.0 [4]) and execute them with respect to an according organization model. Regarding the wide spread of BPMSs, integrating mobile devices into a work environment and its IT infrastructure requires a discussion about the challenges emerging for those.

Pryss, Musiol, and Reichert [5] identified the origin of these challenges in the error-proneness of mobile devices caused by their limited resources and unsound user behavior. A connection loss or an instant shutoff may harm the execution of processes and can hardly be detected by current monitoring tools. To cope with these shortcomings, an automated delegation process and a distinct life cycle for mobile tasks were introduced. This thesis will extend the concepts proposed in [5] with a detailed *user prioritization model* and *offline tasks* to foster the delegation service and improve its flexibility. Based on this, an integration concept for mobile devices into an existing BPMS, using a three layer architecture to decouple the mobile task execution from the underlying BPMS, will be introduced. To proof the feasibility of the integration concept, a prototype using REST[1] services was implemented and validated against real BPMSs.

This work is structured as follows: In Chapter 2, related projects handling the integration of mobility into business processes will be discussed and categorized. Chapter 3 will recap the concepts of prior research work [5] and introduce a *user prioritization model* and *offline tasks*. In Chapter 4, different aspects of integrating mobile task execution into existing business process environments will be discussed. Based on these findings, an integration concept implementing the *mobile task life cycle* [5] will be introduced. Chapter 5 will discuss implementation details of the prototype and will validate it against three real BPMSs. Finally, Chapter 6 will give a conclusion and will show the possibilities for future research work.

[1] Representational State Transfer (REST)

2 Related Work

With the fast improvements of mobile technology in the last century, the importance of mobile and pervasive computing has increased in all disciplines of computer science. The new challenges of mobile computing were already discussed during the nineteen-nineties [6] [7]. Furthermore, the new technologies have set the stage for a whole set of new applications (e.g., medical- [8][3] and business-applications [**barnes2003enterprise**]).

In this chapter, the current research work on introducing mobility into business workflow environments will be presented. Subsequently, a classification model for mobile workflow middleware will be introduced.

The practical approaches which tackle the challenges of combining business-process management and mobile computing can be categorized in two groups: First, new business-process execution languages or extensions for existing ones, which are able to handle mobility in a workflow environment. Second, middleware architectures and frameworks, which use existing paradigms to manage mobility and error handling in a mobile workflow environment.

2.1. Execution Languages and Extensions

Philips, Van Der Straeten, and Jonckers [9] [10] introduced the workflow language *NOW*. Its main focus is on the execution of mobile tasks in decentralized, nomadic networks (e.g., Mobile Ad-hoc Networks (MANETs)). Each participating device has to provide a specific set of services, which are populated into the network by an underlying service discovery mechanism. Furthermore, *NOW* provides a set of high-level control-flow patterns and a mechanism for error handling and resolving at service and network level.

Hackmann, Gill, and Roman [11] described a set of extensions for the process execution language *BPEL* [12], which eliminates shortcomings of *BPEL* in terms of inter-operable pervasive computing. Therefore, partner links which can be bound to multiple endpoints, so called *partner groups*, have been introduced. Moreover, a multicast messaging option is provided and multiple messages can be sent using one partner link. All of these extensions aim to handle an arbitrary number of clients. More importantly, the paper points out, that the amount of actually available devices is unknown, which is a common challenge in mobile computing.

MobWEL [13][14] is an context-aware workflow execution language for mobile processes. It adapts established approaches of context management and content behavior management to define a new language. Contrary to the approaches described before, *MobWEL* takes the context of a mobile device into account. For example, a low battery level can influence the decision whether certain operations may be executed or not.

2.2. Middleware Architectures

DEMAC [15] is a project at the University of Hamburg, trying to realize a service based environment for the execution of distributed business processes. Therefor, different mobile services and messaging protocols are defined to archive a complete decentralized infrastructure. The distributed members (e.g., mobile devices, laptops, or stationary PCs) use the services to exchange messages with the task and process information according to their context. Due to the absence of a central authority (e.g., process engine), the process handling is up to the service infrastructure and the applications themselves. *DEMAC* represents a new approach in workflow management systems. Therefore, a new infrastructure is needed. Interfaces to integrate legacy systems (e.g., classic workflow management systems) are not provided. Furthermore, without a central authority, monitoring and organizing such distributed systems may become a crucial challenge.

MARPLE [16] is a heavyweight approach to execute business process on mobile devices. It implements a process engine on the mobile device itself, partitions the main process and executes this partition on the device. A critical point of this approach is the higher workload for the mobile devices. The engine's footprint has to be minimal to run properly on the limited resources of today's mobile devices. Furthermore, *MARPLE* does not provide an explicit error handling for mobile related failures (e.g., lost connections or broken devices).

Sliver [17] is another workflow engine, running on mobile devices. Its main focus is to provide a minimal footprint. For this reason, it implements a pluggable component architecture with a lightweight XML/SOAP parser using features which J2SE, Java Foundation Profile, and MIDP 2.0 have in common. Hence, it can be deployed on any device which supports any of these standards. Its current footprint is about 114 KB. Additionally, it implements the BPEL extension by Hackmann, Gill, and Roman [11] mentioned earlier.

Rome4U [18] is a commercial solution for a mobile business process management infrastructure. It uses a process engine as a central authority and implements services as communication interfaces. *Rome4U* represents a complete solution with its own process engine and a specialized data format and process representation. Interfaces to integrate standard BPM systems are not provided. Additionally, *Rome4U* uses adaptive processes for escalation and error handling. If errors occur (e.g., a mobile device is not available), the user has to adapt the process to proceed.

The *SAMPROC* project [19], which has been developed at the University of Ulm, has a more technical and generic view. The project tries to offer a framework for realizing distributed, mobile applications and is not particularly focused on business processes. To fulfill this goal, distributed systems are mapped on process flows. Every activity in the control flow represents a node in the distributed system. To handle escalations, adaptive processes are used. For execution, *SAMPROC* uses web-services and an extended BPEL version.

CiAN [20] is a workflow engine designed for MANETs. Due to the nature of such networks, *CiAN* uses a choreography approach instead of an orchestration approach. This means, that it has no central authority. Each node in the network has its own workflow management system instance running. Task specifications are propagated into the network and each node may allocate them. As soon as a predecessor is finished, the next task has to be invoked on the responsible node. *CiAN* provides two modes: First, the planning mode, in which each participating node will be informed about its role in a workflow. Second, the standard mode, in which the execution takes place.

Tuysuz, Avenoglu, and Eren [21] introduced a workflow based guidance framework for managing personal activities. It is designed to support users in every day tasks in a pervasive environment. The framework consists of a mobile application, a workflow management system and a central coordination management system (*CoMS*). Tasks are represented by control flows and executed in the petri net based workflow management system YAWL [22]. The *CoMS* is responsible for processing and delivering messages on a bidirectional communication channel between the server and the mobile devices. Furthermore, it can use additional context information for automated operations. The message interface between *CoMS* and mobile clients assumes that there is always a network connection available. Therefore, no recovering mechanisms are provided. However, the communication is designed to minimize the network usage by using the topic-based publish/subscribe protocol *MQTT* [23].

2.3. Classification of Mobile Workflow Middleware

A first classification of the related work was already achieved by subdividing it into *execution languages* and *middleware frameworks*. Since the integration concept in this works represents a middleware, it is important to define a refined classification model for mobile workflow middleware. Therefore, a set of classification attributes has to be defined. Based on the related work, the following attributes can be identified:

1. The underlying integration paradigm
2. The underlying infrastructure
3. How the mobile process execution is implemented
4. The degree of integration dimensions

With these attributes, mobile workflow middleware solutions can be set into perspective and hence become comparable. The attributes will subsequently be discussed in more detail.

2.3.1. Classification Attributes

Integration paradigm: It is important to define what type of integration an approach follows. Basically, there are three integration paradigms in terms of mobile process support:

1. Integration of mobile task into an existing business process environment
2. Introduction of workflow support for a specific infrastructure (e.g., a MANET)
3. Use of mobile tasks to implement a distributed environment

The underlying integration paradigm constitutes an approach's focus. Approaches with distinct paradigms are often not compatible because of different preconditions and requirements.

Infrastructure: This property describes the communication structure of the target infrastructure. In Sen, Roman, and Gill [20], this was categorized into *orchestration* and *choreography* (cf. Figures 2.1 A and B).

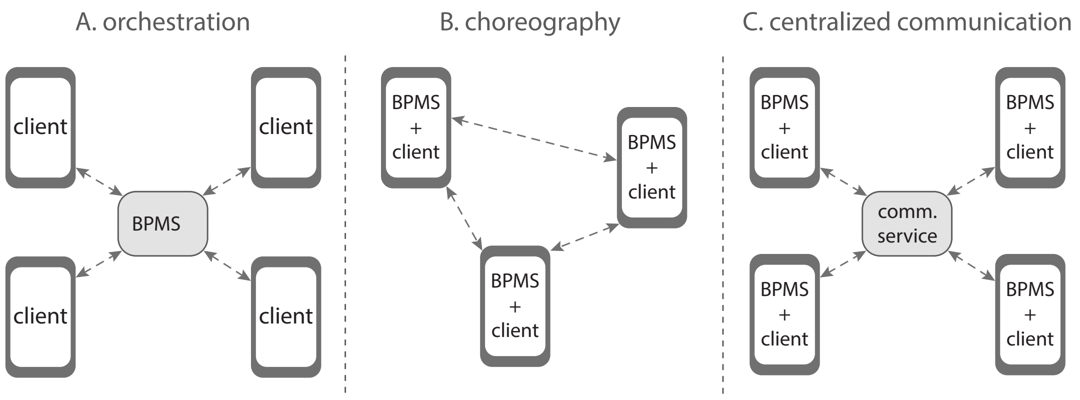


Figure 2.1.: Communication Infrastructures

- *An orchestration infrastructure* has a central authority. It represents the workflow management system which is responsible for task allocation and invocation.
- *A choreography infrastructure* relies on a decentralized infrastructure, using network propagation- and service discovery protocols to allocate and invoke mobile tasks.

Furthermore, there is the possibility of a central service which does not orchestrate the workflow, but serves as an communication interface instead (cf. Figure 2.1 C), as seen in Kunze [15]. This can be called a *choreography infrastructure with a centralized communication service*.

Implementation of mobile process execution: In [5], three approaches of realizing mobile process execution were introduced. The approaches are shown in Figure 2.2:

1. *Physical process fragmentation:* A process (i.e., process schema) is physically partitioned during design time. The resulting process fragments and their tasks are then assigned to a number of mobile devices before run time.
2. *Logical process fragmentation:* A process schema is partitioned logically. In this case, the resulting process fragments and their tasks are executed on different mobile devices. Opposed to the first approach, the original process schema will be preserved during run time when executing the process fragments. Usually, migration techniques are applied in this context (Zaplata et al. [24]); e.g., based

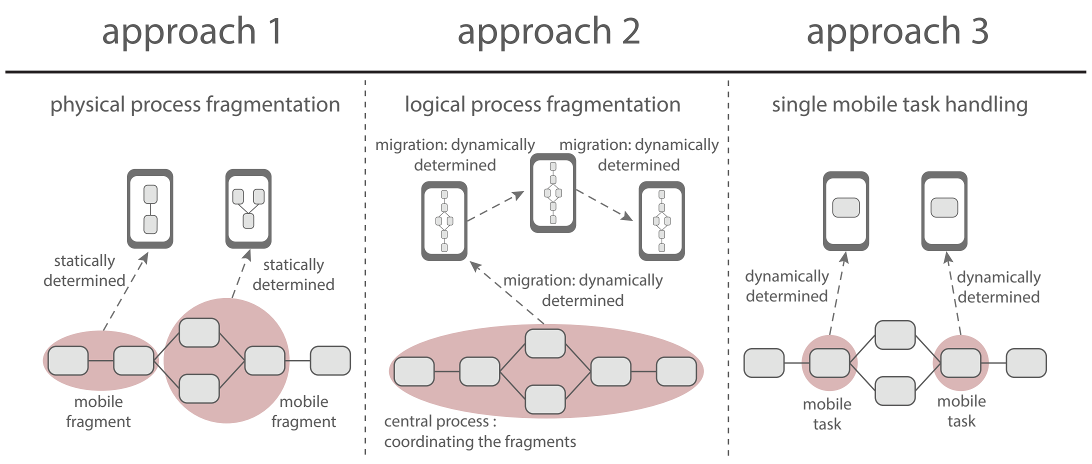

approach 1
approach 2
approach 3
physical process fragmentation
statically determined
statically determined
mobile fragment
mobile fragment
logical process fragmentation
migration: dynamically determined
migration: dynamically determined
migration: dynamically determined
central process : coordinating the fragments
single mobile task handling
dynamically determined
dynamically determined
mobile task
mobile task

Figure 2.2.: Approaches for realizing mobile task execution

on the original process schema, it can be determined how the migration between logical process fragments is to be accomplished at run time. Accordingly, the device to execute each process fragment is determined dynamically. This allows, in particular, dynamic exchanges of devices already assigned to a fragment.

3. *Single mobile task handling:* Single process tasks are executed on mobile devices. For this purpose, a mobile device must cover a subset of a stationary process client's functionality; e.g., a worklist component that is continuously updated by the process engine.

Degree of integration: This attribute describes, whether an approach provides interfaces for integrating existing infrastructures (e.g., other BPMSs or user repositories). Due to the rather abstract nature of this attribute, further values have to be defined:

- **High:** The architecture provides well documented interfaces for integration purposes.
- **Medium:** The architecture provides no interfaces for integration purposes, but it is possible to integrate other technologies by implementing own interfaces or extensions.
- **Low:** Integration is not intended.

2.3.2. Classification of Related Approaches

Based on the classification model, the related work can be classified as shown in Table 2.2. To improve readability, attributes are denoted with the keys introduced in Table 2.1.

Attribute	**①**	**②**	**③**
Integration paradigm	Mobile tasks into existing BPMS	Introducing workflow support	Implementation
Infrastructure	Orchestration	Choreography	Centralized communication
Implemen-tation	Physical process fragmentation	Logical process fragmentation	Single mobile task handling

Table 2.1.: Keys for Table 2.2

Project	**Integration paradigm**	**Infra-structure**	**Implemen-tation**	**Degree of integration**
DEMAC	②	③	②	medium
MARPLE	①	③	①	medium
Sliver	②	②	②	low
Rome4U	①	①	③	low
CiAN	②	②	②	low
SAMPROC	③	③	②	high
Mobile guidance framework	③	①	③	low

Table 2.2.: Classification of related projects

2.4. Further Aspects of Mobile Task Execution

Wakholi and Chen [25] discussed the challenges of process fragmentation in a mobile workflow environment under the assumption of a unstable network connection. They argue, that in a centralized workflow environment, the fragmentation of processes into groups to execute them on disconnected mobile clients, needs special treatment which provides an option that enables the server to maintain control. Therefore, additional workflow partitioning rules on top of existing fragmentation approaches have been introduced. They are based on structural and behavioral aspects of the original process model. Furthermore, an automatic partitioning algorithm has been developed, which uses the fragmentation rules to discover valid fragments.

The approach of Hahn and Schweppe [26] copes with correctness in mobile service environments. It argues, that the strict mechanisms of traditional systems (e.g., database management systems) are not sufficient in highly flexible infrastructures, since they would lead to blocking states. Hence, a relaxed atomicity for composite services based on transactional service properties is introduced. Composite Services, which are designed as control flows of single services, provide attributes such as *compensatability*, *redoability* and *recoverability*. Furthermore, the designer can declare multiple sets of services, whose completion reflect the successful execution of a composite service. In this way, the services can be dynamically adapted, based on the current execution context. After the dynamic service adjustment, an automatic recovery mechanism is used to satisfy all service dependencies and guarantee a correct completion.

3 A Framework for Mobile Task Integration

Providing mobility in a workflow infrastructure implies new challenges caused by the nature of mobile technologies. To be able to handle these challenges and enable a proper integration of mobile tasks into a process environment, our prior research work [5][27] proposed a framework which forms the foundation of an integration concept for mobile support in BPMSs. It introduces a set of operations that defines a life cycle for mobile tasks, and uses a delegation and backup approach to avoid deadlocks in mobile workflows, caused by network errors or malicious user behavior. For example, if a user goes offline while executing a mobile task, the task is automatically delegated to the next best matching mobile user. If there is no appropriate user available, the task will go into a backup state, in which it can be executed by a stationary client. Since a stable network connection is expected in this scenario, the backup operation is considered as an exception and not a desirable way to perform a task.

This chapter will recapitulate the framework's main concepts and extend them at some point. In Section 3.1, the main challenges resulting from a mobile context will be recapped and evaluation models for location and user behavior will be introduced. In the following sections, the framework's main operations will be discussed. Section 3.2 explains how a mobile task can be declared and in Section 3.3, pre-filters will be introduced. The main concepts, the *Mobile Delegation Service (MDS)* and the backup service will be reviewed in Sections 3.4 and 3.5. Additionally, a prioritization model for the user list management and the concept of offline tasks will be added. Finally, Section 3.6 will summarize the chapter by introducing a life cycle for mobile tasks.

3.1. Challenges for Executing Processes with Mobile Tasks

The introduction of mobility to a business process environment imposes a set of new challenges which has to be dealt with properly. On one hand, this is caused by the fast changing context of a mobile environment (e.g., changes in location or count of participating devices). This changing context is often handled by context-aware systems [28] [13]. Four context related challenges can be identified, which are either caused by the characteristics of a mobile environment or by the users' behavior. On the other hand, challenges emerging from the process infrastructure have to be considered. In total there are seven challenges, described below:

Challenge 1: Connectivity (Environment)
Connectivity refers to the availability of users and the mobile devices assigned to them. In turn, unavailability might be caused by an undesired status of a device (e.g., broken device) or a specific personal status (e.g., user is on vacation). Finally, a mobile device will only be considered as a target device for executing mobile tasks if it is connected to a network.

Challenge 2: Low Battery (Environment)
A mobile device with a low battery status should not be considered as target platform for executing a mobile task until its battery has been recharged; i.e., a low battery status indicates that the device (and its user) shall not be considered at the moment.

Challenge 3: Instant Shutdown (User Behavior)
In practice, users might instantly shut down their mobile device without reflecting on the consequences. This usually constitutes a short-term problem and the device can be restarted soon in most cases. If a user exhibits many instant shutdowns, however, this misbehavior needs to be considered. The presented approach maintains the numbers of the instant shutdowns applied.

Challenge 4: User Location (User behavior)
At run time, attribute *UserLocation* maintains the current location of a mobile user. If the latter is to execute a mobile task at a location different from the present one, this needs to be considered.

Challenge 5: Data Consistency (Process)
Data dependencies between process activities result from the order in which activities read and write process data objects. In the presented approach, mobile tasks providing data for other tasks are specially treated in order to ensure data consistency in case of task failures.

Challenge 6: Location (Process)
Each mobile task has an attribute *Location* that optionally stores the location where this task shall be performed. Note that in certain cases, data or physical objects needed to accomplish a task are only available at a certain location. If the user is performing her work while continuously moving, it cannot be guaranteed that she is on the right spot to gather the data needed.

Challenge 7: Urgency (Process)
The urgency of a mobile task needs to be considered as well. For example, if a lab test is required in the context of an emergency surgery, the urgency of the task performing this lab test will be high. The value of the respective mobile task attribute either is null or describes the point in time the task shall be performed, i.e., either a concrete point in time or a period. In the latter case, the mobile task must be finished within the specified period after having been allocated it to a mobile user.

3.1.1. Location Representation in Mobile Environments

When discussing mobile environments, location is a crucial factor because of its variability and alteration rate. Challenges 4 and 6 take this into account by defining the *task location* and the *user location*, but this is not enough for practical use. Therefore, a well defined location model is needed. Becker and Dürr [29] identified two basic categories of location models:

1. **Symbolic location models:** The location can be represented in a symbolic domain (e.g., Coordinates).
2. **Geometric location models:** The location is represented by geometric figures, describing a location (e.g., a circle representing an area). The figures have to be transferable into a symbolic domain.

To provide a complete location model for mobile business process environments, the properties of mobile tasks, which are affected by these models, have to be discussed. In particular, these are *where* a task can be executed, which is the equivalent to a task position, and *who* may execute the task, which needs an evaluation of the user position related to a task position. Based on these assumptions, a task can be executed only in an exact position (e.g., exact coordinates) or in a region, surrounding an exact location (e.g., in a radius around exact coordinates). Furthermore, it can be only executed by persons who are at the task's location (e.g., inside the radius) or nearby (e.g., within a certain radius surrounding the task location).

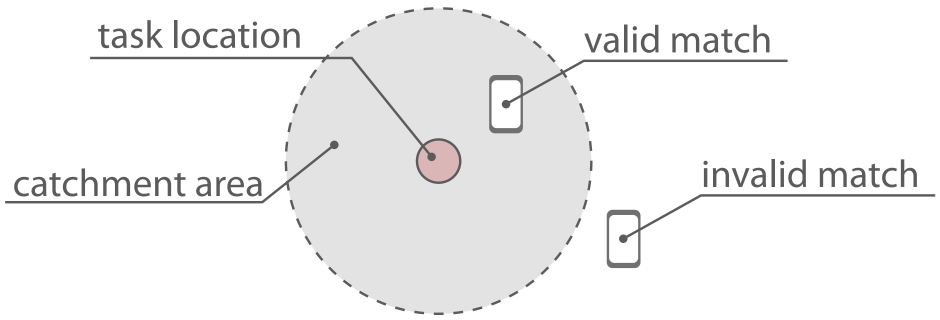


Figure 3.1.: Location model for mobile tasks

Therefore, the attributes *task location* and *user location* can be defined as follows:

- **Task location** is represented by an *Execution Location*, which can be the position or region where the task needs to be executed, and by a *Catchment Area* within which users are still considered as a valid match (cf. Figure 3.1).
- **User location** is represented by an exact position in a symbolic location model.

Additionally, the model has to provide a possibility to compare distinct matching user locations. This can easily be done by calculating and comparing distances between the *task location* and all matching *user locations*. Therefore, distance $dist_{MU}$ is defined as $dist_{MU} = loc_T - loc_{MU}$ where loc_T is the task location and loc_{MU} is the position of a distinct mobile user, who is entitled to execute the task. To make the model usable in a system with more variables, the distances have to be normalized. Thus, the *normalized location factor (* nlf_{MU}*)* for a mobile user MU is introduced. In the following, let loc_{CA} be the catchment area and $dist_{maxCA}$ the distance between loc_T and the furthermost border of this area. Then nlf_{MU} can be written as:

$$nlf_{MU} = \begin{cases} \frac{dist_{maxCA}}{dist_{MU}} = \frac{dist_{maxCA}}{loc_T - loc_{MU}} & \forall MU \in loc_{CA} \\ 0 & \forall MU \notin loc_{CA} \end{cases} \tag{3.1}$$

3.1.2. Measuring and Evaluating User Behavior

Next to location, Challenge 3 identifies *instant shutdowns* as the second important user related factor in mobile networks. It proposes a single counter to track and evaluate the users behavior. Since a counter shows quantity only and does not give any information about the quality (e.g., how long was a device offline), this approach is not sufficient. Furthermore, other aspects of user behavior, for example how does a user perform tasks or where are common locations for a user, have not been discussed yet.

Location dependent behavior: Since in most mobile scenarios, the location of mobile devices is constantly tracked, it can be used for long-term evaluations and system optimization as well [30] [31] [32] [33]. Location-based behavior patterns [34] can thus be used to analyze a user's behavior during design time and adapt a process according to these patterns. For example, a user who is rarely in the location in which a task has to be performed may not be assigned to this task.

Unsound user behavior: Another important impact on the overall robustness of mobile task execution is unsound user behavior. This includes inappropriate device handling like instant shutdowns or running the device constantly on low battery. But actions directly related to task execution, for example, taking over tasks, but constantly running into the time-out, also have to be considered. It is important to determine whether an event is caused by unsound user behavior or by technical issues. Accordingly, different cases have to be discussed, especially as relates to connectivity. Three different reasons for a connection loss can be identified:

- Device issues (e.g., broken device, low battery)
- Network issues (e.g., no WiFi signal)
- User issues (e.g., unexpected instant shutdowns)

Device issues are easy to identify, by constantly advertising a device's technical status (e.g., battery status and battery draining). On the contrary, to determine if a connection loss was caused by network issues or because of a user action, is only trivial to handle, as long as a standard procedure to shut down the device was used. However, if someone just pulls out the battery, this is hard to distinguish it from a normal network connection loss. A solution to this could be to not just advertise the battery status, but also the used connection and the signal strength. On the other hand, the long term impact of both, network issues and recurring instant shutdowns, is the same: it delays and compromises the execution of a mobile task. Since network issues are mostly caused by the user as well (e.g., leaving the network area), these two aspects should be handled equally. Furthermore, the time a user is usually offline after losing the connection, and the average time, a device is run on low battery, have to be taken into account as well. Based on the prior assumptions, it is possible to define a model to measure unsound user behavior. Accordingly, two different *user behavior factors* have to be introduced for a mobile user MU. First, the *device behavior factor* (dbf_{MU}), which represents the degree of device issues by taking the number of connection losses, the average time of a connection loss and the average time a user is running his device on low battery, into account:

$$dbf_{MU} = \begin{cases} \frac{1}{\text{\#conn. losses} + \varnothing\text{conn. loss time} + \varnothing\text{low battery time}} & \text{if \#conn. losses} \neq 0 \\ 1 & \text{if \#conn. losses} = 0 \end{cases} \quad (3.2)$$

It follows that if $\#conn.\ losses \in \mathbb{N}\backslash\{0\}$, then $0 < dbf_{MU} < 1$ holds.

The second factor is called *execution behavior factor (ebf_{MU})*, which utilizes a users task execution behavior based on the number of started and delegated tasks:

$$ebf_{MU} = \begin{cases} \frac{\text{\#tasks started by MU}}{\text{\#tasks delegated by MU}} & \text{if \#tasks delegated by MU} \neq 0 \\ 1 & \text{if \#tasks delegated by MU} = 0 \end{cases} \quad (3.3)$$

3.2. Declaration of Mobile Tasks

Mobile tasks have to be modeled by the designer when creating a process. Therefore, a set of attributes, considering the challenges in Section 3.1, has to be added (e.g., task location and urgency). Furthermore, a threshold, representing the minimal amount of active mobile users for delegation, is needed. This operation is called *mobile process transformation* (cf. Figure 3.2 ①).

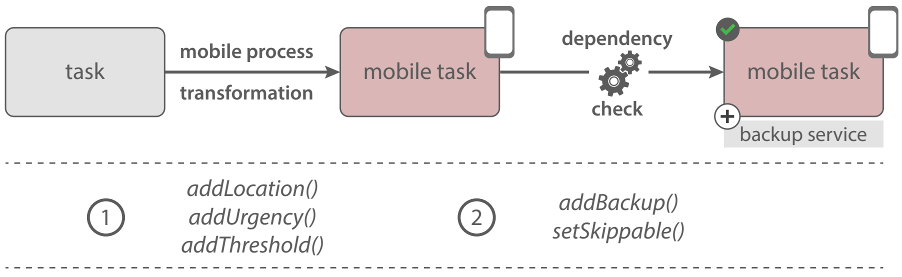


Figure 3.2.: Declaration of mobile tasks

Following the assumption that exception handling is only mandatory if a mobile task has to satisfy data dependencies, an automated dependency check appends a backup service to the task if necessary. Otherwise it will be marked as optional, so that it can be skipped during run time. This can be changed manually by the designer as well (cf. Figure 3.2 ②).

3.3. Pre-Filters

To provide more flexible user configuration abilities on the process instance level, user filters can be applied when a mobile process gets initiated. Therefore, an additional filter-list with mobile user entries is added to a mobile task (cf. Figure 3.3). The users on the lists will then be ignored in the user list calculations during run time. For example, if

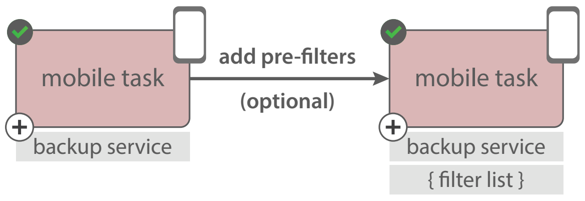


Figure 3.3.: Adding pre-filters to a mobile task

a supervisor initiates an every day task and knows that some workers are exclusively assigned to another project, he can define pre-filters targeting them so they will not be considered for this process instance during execution.

3.4. Mobile Delegation Service

While the mobile task declaration takes place during design time, the *Mobile Delegation Service (MDS)* manages the execution of mobile tasks at run time by using an automated delegation approach, which ensures, that already assigned mobile tasks are automatically re-delegated to another authorized mobile user in the case of errors. Compared to the traditional *user-to-user* interaction pattern, where a user explicitly transfers rights to another user [35], the delegation service follows a *system-to-user* pattern. Therefore, the system enforces a task delegation as soon as an execution error occurs (cf. Figure 3.4).

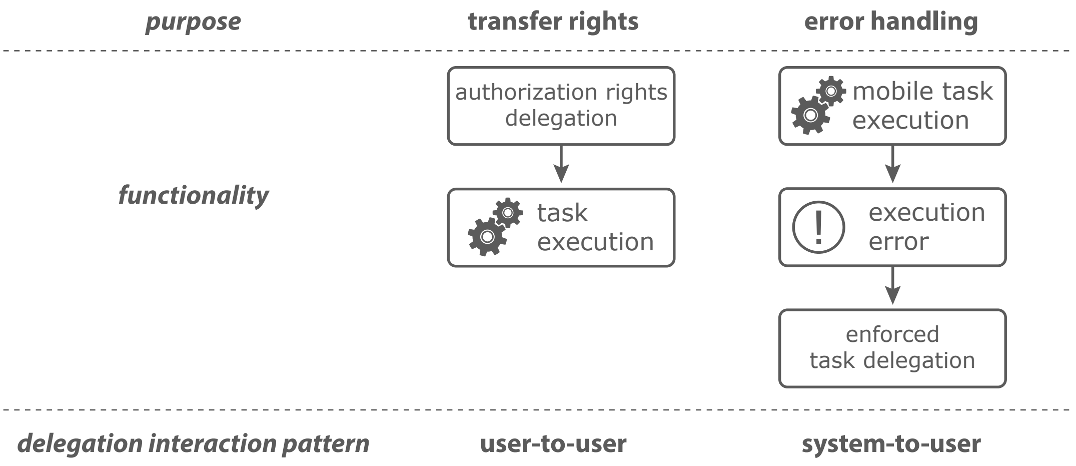


Figure 3.4.: Different delegation mechanisms

3.4.1. User List Management

The *MDS* has to maintain three different user lists to guarantee a robust execution of mobile tasks: an initial user list ul_{init}, a mobile user list ul_{mob}, and a delegation list dl_{mob}. ul_{init} containing all mobile users u_{mob} authorized to execute a mobile task t_{mob} and is provided by the process engine (cf. Figure 3.5).

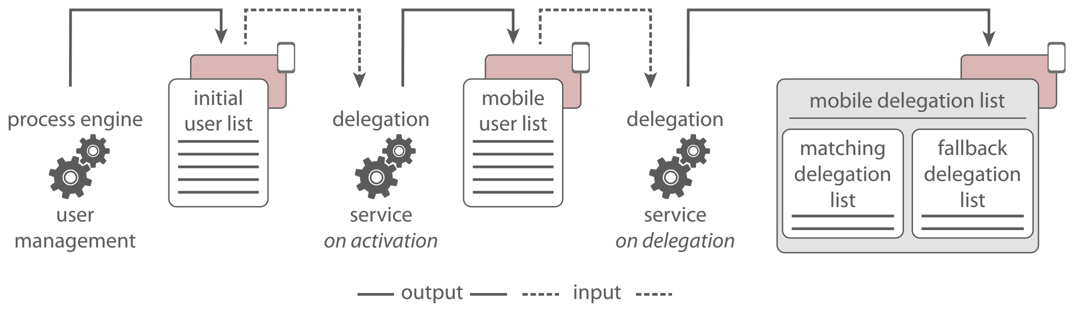


Figure 3.5.: User list management

Mobile user list calculation: Based on ul_{init}, the *MDS* calculates ul_{mob} by evaluating the connectivity, location and battery status of each user's device in ul_{init}. The location is represented by the location model proposed in Chapter 3.1.1. The requirements for a mobile user u_{mob}, to be regarded as a proper member of dl_{mob} for a mobile task t_{mob} are:

1. u_{mob} is currently online. $\rightarrow u_{mob}.connectivity = true$
2. The user location of u_{mob} is within the catchment area of t_{mob}. $\rightarrow u_{mob}.loc \in t_{mob}.loc_{CA}$
3. The current battery status is not critical. $\rightarrow u_{mob}.lowBattery = false$
4. u_{mob} is not subject to pre-filter. $\rightarrow u_{mob}.preFilter = false$

The user behavior factors dbf and ebf from Chapter 3.1.2 can also be used as criteria by evaluating thresholds th_{dbf} and th_{ebf}. Hence, the following optional requirements have to be added to the list above:

5. $dbf_{u_{mob}} \geq th_{dbf}$
6. $ebf_{u_{mob}} \geq th_{ebf}$

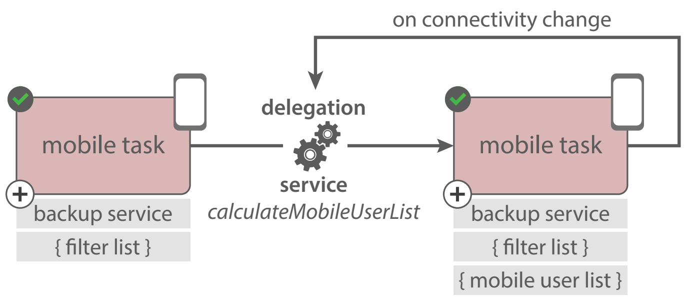


Figure 3.6.: Calculating the mobile user list

As soon as a mobile task is activated or the connectivity of a mobile user in ul_{init} changes while the task has not been started or delegated yet, the *MDS* calculates ul_{mob} by applying the following procedure (cf. Figure 3.6):

```
procedure CALCULATEMOBILEUSERLIST($t_{mob}$ , $ul_{init}$)
    $ul_{mob} \leftarrow \{\}$
    for all $u_{mob}$ in $ul_{init}$ do
        if $u_{mob}.connectivity$ and $\neg(u_{mob}.lowBattery)$ and $\neg(u_{mob}.preFilter)$ then
            $evalBehavior \leftarrow (dbf_{u_{mob}} \geq th_{dbf}$ and $ebf_{u_{mob}} \geq th_{ebf})$
            $evalLoc \leftarrow TRUE$
            $isLocSet \leftarrow t_{mob}.loc \neq \emptyset$
            if $isLocSet$ then $evalLoc \leftarrow u_{mob}.loc \in t_{mob}.loc_{CA}$
            if $evalLoc$ and $evalBehavior$ then
                $ul_{mob}.append(u_{mob})$
```

Each time, the procedure is called, it will empty ul_{mob} and re-add each mobile user in ul_{init} who matches the requirements mentioned before.

Prioritization model for delegation lists: While ul_{mob} is an unsorted list, dl_{mob} is used to determine the best fitting user for an upcoming delegation and therefore has to be prioritized. So far, a static prioritization based on the *battery status* and an *instant shutdown counter*, which records the user's instant shutdown behavior, has been proposed. A more flexible approach is a prioritization model which can be used to calculate a normalized numeric priority value $Pr_{u_{mob}}$ for each mobile user u_{mob}. For the calculation, the following factors have to be taken into account:

- User location (nlf)
- Unsound user behavior (dbf and ebf)
- Location dependent behavior by using location based behavior patterns ($lbbp$)
- Constraint related prioritization factors (cpf)
- User defined prioritization factors (upf)

The used $lbbp$ can be chosen freely. The only condition is that it has to be represented in a single numeric value. upf is a user specific prioritizing factor defined by the designer and can be used to down- or upgrade specific users individually. cpf is defined by the constraint management to foster the mobile execution with entailment constraints [27]. Since different scenarios may have various requirements, a static combination of all factors is not sufficient. Therefore, a weighting has to be provided.

Finally, the following model can be used to calculate a priority $Pr_{u_{mob}}$ for a mobile user u_{mob}:

$$Pr_{u_{mob}} = (a \cdot nlf) + (b \cdot dbf) + (c \cdot ebf) + (d \cdot lbbp) + (e \cdot cpf) + (f \cdot upf) \qquad (3.4)$$

It is imperative that if $Pr_{u_{mob1}} > Pr_{u_{mob2}}$, then u_{mob1} has a higher priority than u_{mob2}.

Delegation list calculation: As soon as a delegation is pending, the *MDS* refreshes ul_{mob} and calculates dl_{mob} (cf. Figure 3.7). To enhance flexibility, dl_{mob} is subdivided into the *matching delegation list* dl_{match} and the *fallback delegation list* dl_{fb}.

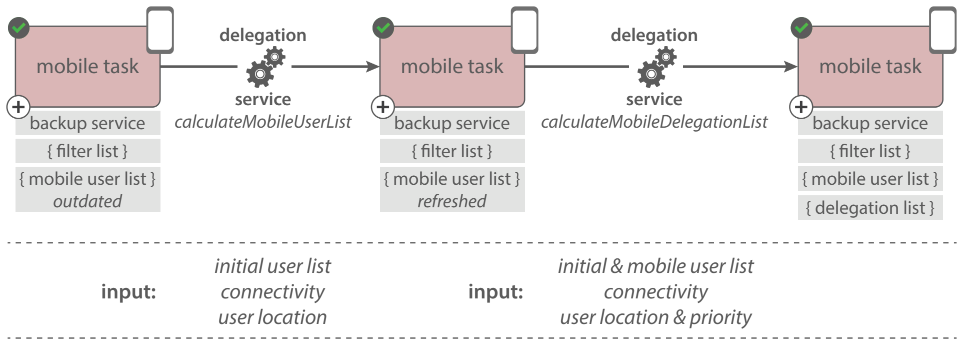


Figure 3.7.: Calculating the mobile delegation list

The requirements for dl_{match}, which is used as the main look-up repository, are the same as for ul_{mob}. Additionally, dl_{fb} contains the users, who matches all requirements but the location. If dl_{match} runs out of members, users outside of the task's catchment area can thus be used as delegation target. Moreover, the battery status is used as a prioritization factor only and not as a disqualification criterion. The following procedure implements the calculation process.

```
**procedure** CALCULATEMOBILEDELEGATIONLIST($t_{mob}$ , $ul_{mob}$)
    $dl_{match} \leftarrow (), dl_{fb} \leftarrow ()$
    $dl_{mob} \leftarrow (dl_{match}, dl_{fb})$
    **for all** $u_{mob}$ **in** $ul_{mob}$ **do**
        $evalBehavior \leftarrow (dbf_{u_{mob}} \geq th_{dbf}$ **and** $ebf_{u_{mob}} \geq th_{ebf})$
        **if** $u_{mob}.connectivity$ **and** $\neg(u_{mob}.preFilter)$ **and** $evalBehavior$ **then**
            $evalLoc \leftarrow TRUE$
            $isLocSet \leftarrow t_{mob}.loc \neq \emptyset$
            **if** $isLocSet$ **then** $evalLoc \leftarrow u_{mob}.loc \in t_{mob}.loc_{CA}$
            **if** $evalLoc$ **then**
                $dl_{match}.put(u_{mob}, Pr_{u_{mob}})$ [
            **else**
                $dl_{fb}.put(u_{mob}, Pr_{u_{mob}})$ ]
```

3.4.2. Service Execution Flow

While executing a mobile task, the *MDS* assigns different states to it. Tasks may enter seven distinct states, denoted as t(*<STATE>*), and the respective transitions as T_x (cf. Figure 3.8).

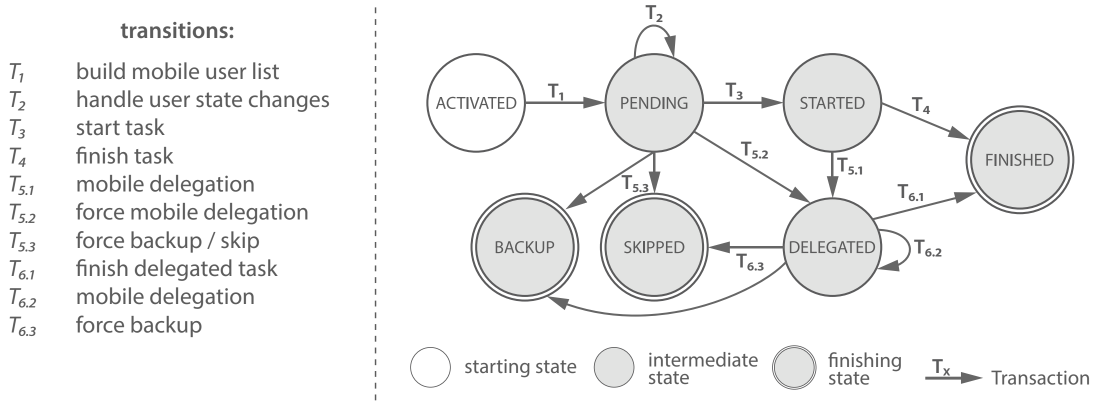


Figure 3.8.: MDS execution flow

The execution flow starts, as soon as the task is initiated and therefore enters the state *ACTIVATED*. After creating ul_{mob} (T_1), the task enters *PENDING*. From here on, ul_{mob} may be recalculated because of user base changes, hence it stays *PENDING*(T_2). At some point, the execution flow has to continue to ultimately end in one of three finishing states:

- *FINISHED:* the task has been finished correctly.
- *SKIPPED:* the task has to be skipped.
- *BACKUP:* the task needs a backup.

Therefore, taking an urgency to_u ($to_u = 0$ denotes a timeout) and a user list threshold th_{mul} as given, the following four execution flows are possible:

Normal task execution:

$user_a \in ul_{mob}$ starts mobile task t and performs it.

$t(PENDING) \rightarrow T_3 \rightarrow t(STARTED) \rightarrow T_4 \rightarrow t(FINISHED)$

Delegated task execution:

$user_a \in ul_{mob}$ starts mobile task *t*. Then she goes offline. *t* will now automatically be delegated to another user $user_b \in ul_{mob}$, who finally finishes the mobile task.

$T_3 \rightarrow t(STARTED) \rightarrow T_{5.1} \rightarrow t(DELEGATED) \rightarrow T_{6.1} \rightarrow t(FINISHED)$

Forced delegation:

A forced delegation becomes necessary if the task is pending and $|ul_{mob}| < th_{mul}$, or $to_u = 0$. Additionally, if the task has already been delegated to $user_b$, whose state changes to *offline*, *t* must be delegated to another user $user_n \in ul_{mob}$.

$t(PENDING) \rightarrow T_{5.2} \rightarrow t(DELEGATED)$
$t(DELEGATED) \rightarrow T_{6.2} \rightarrow t(DELEGATED)$

Skip or Backup:

If *t* is pending and ul_{mob} is empty or if the task is delegated and dl_{mob} is empty, a skip or backup will be performed.

$t(PENDING) \rightarrow T_{5.3} \rightarrow t(SKIPPED) \vee t(BACKUP)$
$t(DELEGATED) \rightarrow T_{5.3} \rightarrow t(SKIPPED) \vee t(BACKUP)$

3.4.3. Introducing Mobile Offline Tasks

So far, an environment with constant network connectivity is implied. For instance, as soon as a user takes responsibility for a mobile task, he has to stay connected until the task has been completed. Otherwise, it will be delegated to another user. This behavior is not always desirable, namely, if the network coverage in an area is not very good or if mobile tasks are used by field crews. Hence, the introduction of *Mobile Offline Tasks* is necessary.

Mobile Offline Tasks represent mobile tasks, which will not be delegated if the performing user's state changes to offline. Therefore, one can still perform a task while being offline and only has to come online again to transfer the collected data to the BPMS and to finish the task. The ability to perform tasks offline does, however, weaken the stability of the mobile execution, since the *MDS* will be bypassed. Therefore, to avoid deadlocks and set the *MDS* back in place, an urgency to_u has to be set for each *mobile offline task*. Figure 3.9 shows the semantic of an offline task with defined urgency.

If a mobile user $user_a$ goes offline while performing a mobile task t_{mob}, the task will not be delegated until $to_u = 0$ holds. If one comes back online before this, t_{mob} can be finished

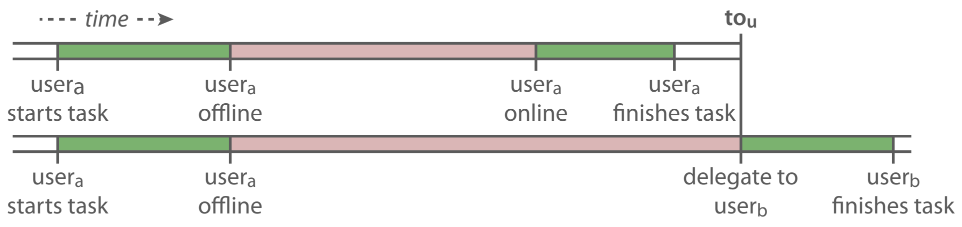


Figure 3.9.: Offline task semantics

regularly by $user_a$. Otherwise, it will be delegated to another mobile user $user_b$. Once the task has been delegated, it will be handled as a regular mobile task. Hence, if $user_b$ goes offline, t_{mob} will be delegated, skipped or backed up. Accordingly, the execution flow of a *delegated task execution* has to be complemented by this requirement.

3.5. Escalation Handling: Backup Service

While the *MDS* fosters the execution of mobile tasks by introducing an automated delegation from one mobile device to another, it needs a minimum number of online mobile users. If this cannot be guaranteed anymore, deadlocks become unavoidable, resulting in a crucial decrease in execution stability. To handle such a worst case scenario, the *backup service* is introduced as an escalation handling for the *MDS*. With this service, a mobile task can be performed on a stationary client as a last resort, if there are not enough mobile users available to complete this task in a mobile manner (cf. Figure 3.10).

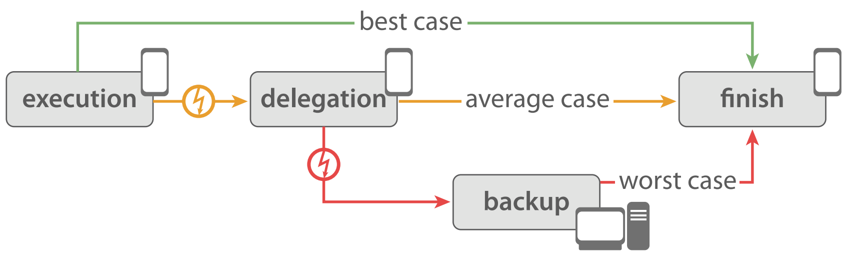


Figure 3.10.: Best, average and worst case scenario for mobile task execution

According to the challenges in Chapter 3.1, the backup service is mandatory if a mobile task has to satisfy data dependencies. If this is not the case, the task can simply be skipped. However, the process designer is still able to add the backup service manually. Furthermore, an optional validation procedure can be applied, to ensure the validity of the provided data.

Basically, the service consists of two operations, which are added to a process fragment replacing the mobile task in case the aforementioned exceptional situation occurs. The first one is called simple backup operation while the second is called complex backup operation. This will be followed by a discussion on how they are implemented and in which context they are applied.

Simple Backup Operation: During design time, all mobile tasks producing data for other tasks are determined. Each of these tasks is then, in turn, automatically associated with a simple backup operation by applying the following steps: If a backup operation is needed for a mobile task B1, the latter is substituted by the process fragment depicted in Figures 3.11 and 3.12. During run time, the execution of backup task B2 on a stationary

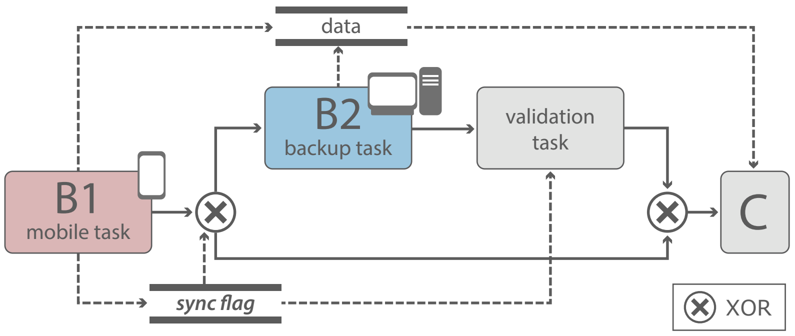


Figure 3.11.: Simple backup operation

computer will then guarantee that subsequent tasks of B1 will not be affected by a failure of this mobile task, i.e., backup task B2 will provide the same data as mobile task B1. In this context, a sync flag guarantees that B2 will be only executed if mobile task B1 fails (cf. Figure 3.11). B1 thus writes the sync flag according to its execution state. If B1 has been executed correctly, the sync flag is set to *true*, otherwise it will be set to *false*. Depending on the respective value, the succeeding XOR process fragment will then be executed as follows: If the sync flag is *false*, the upper branch will be chosen and B2 will be executed.

In turn, if the sync flag is *true*, the lower branch will be chosen and nothing happens; i.e., B2 will only be executed if B1 fails. As shown in Figure 3.11, the simple backup operation comprises another task, i.e., the *validation task*. It is used to manually confirm the execution of B2. The following action will therefore be performed during run time, if the sync flag is set to true and was assigned to the *validation task* during design time: The mobile user responsible for handling the failed mobile task will have to confirm that the backup task has been completed correctly.

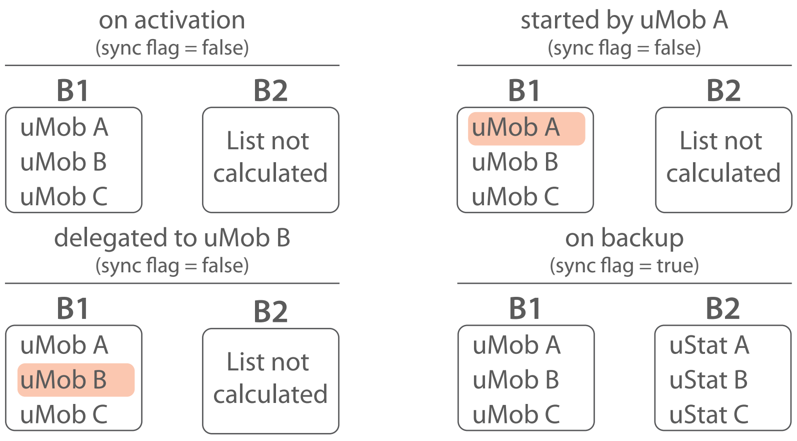


Figure 3.12.: User lists during the simple backup operation

Complex Backup Operation: The complex backup operation shown in Figures 3.14 and 3.13 is provided to deal with urgent mobile tasks. With this operation, backup task B2 can be performed more quickly, based on two changes in comparison to the simple backup operation described above.

First, a userlist task is added. The backup task is then executed in parallel to the mobile task. In order to perform B2 more quickly, the complex backup operation works as follows: First, the user list task determines the lists of authorized users for B1 and B2 respectively (cf. Figure 3.13, on activation). Then, at the time B1 is started, B2 is started synchronously. Following this, task B2 will be locked for all users from the user list of B2. After assigning B1 to a user (cf. Figure 3.13, started by uMob A), the

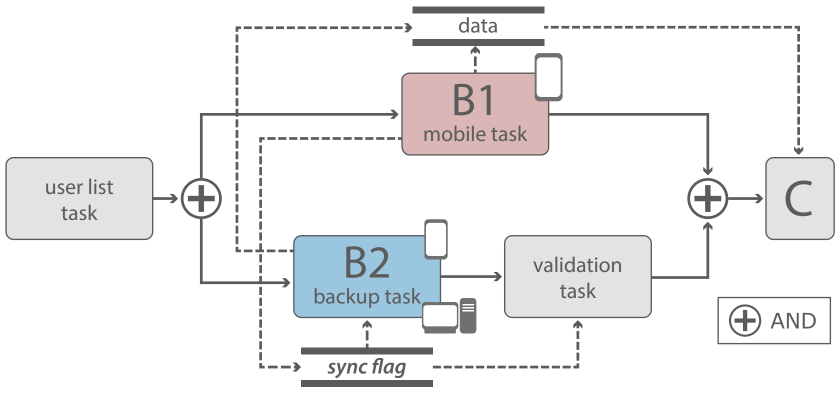


Figure 3.13.: Complex backup operation

user list will be adapted for both tasks. Note that the user list for B2 assigns the task to the same user who performed it on the mobile device as a mobile task. Applying this procedure offers advantages in many respects: First, all other users who may perform B2 are able to monitor which mobile user is currently working on this task. Second, if no other authorized mobile users are available to B1 for delegation, the user list for backup task B2 has been already determined concurrently. Compared to the simple backup operation, for which the user list of B2 is only determined when B1 has been finished, this procedure speeds up user assignment.

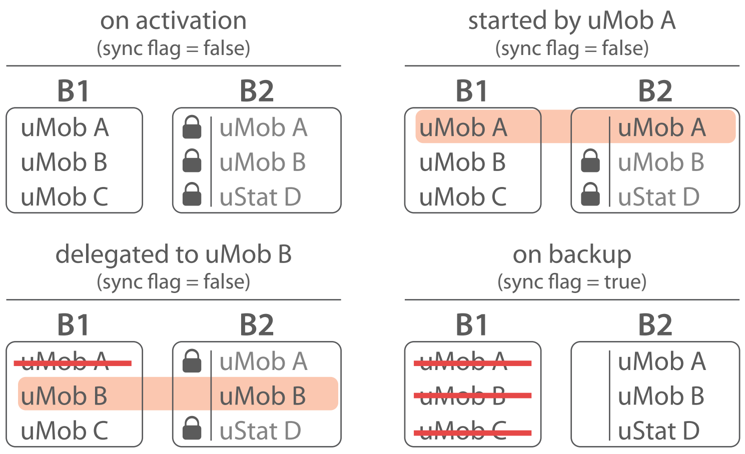


Figure 3.14.: User lists during the complex backup operation

3.6. Mobile Task Life cycle

Based on the operations and concepts discussed in this chapter, a complete life cycle can be defined for mobile tasks, covering design time, instantiation time and run time (cf. Figure 3.15). During a full life cycle, a mobile task can possess nine distinct state:

S1 **Mobile Task Transformation:** This state implies the transformation of a standard task into a mobile task and the calculation of necessary process flow changes (i.e. adding a backup operation) as shown in Section 3.2.

S2 **Pre-Filter Definition:** During *Instantiation Time*, pre-filters can be defined as shown in Section 3.3.

S3 **Task Activated:** As soon as a mobile task is initiated and enters run time it will be considered as activated. It will stay in this state until ul_{mob} is generated by the *MDS*. The state is equivalent to the *MDS* state *ACTIVATED*.

S4 **Task Pending:** This state may be entered as soon as ul_{mob} is generated and is equivalent to the *MDS* state *PENDING*.

S5 **Task Running:** A mobile task may enter this state as soon as it is started by a mobile user. It is equivalent to the *MDS* state *STARTED*.

S6 **Task Delegated:** This state implies, that there is an ongoing delegation process. It is equivalent to the according *MDS* state.

S7 **Task Backed up / Skipped:** In this state, the mobile task has to be backed up or skipped, depending on the task context. This state groups the *MDS* states *BACKUP* and *SKIPPED*. Next, the task has to enter state *S8 Task Completed*.

S8 **Task Completed:** This state marks the end of task execution. At this point, all provided data has to be written or discarded, depending on the final execution status and the task's settings. Furthermore the task has to be marked as *FINISHED*, *BACKUP* or *SKIPPED*, to allow process execution to continue.

S9 **Process Completed:** This state marks the end of process instance execution.

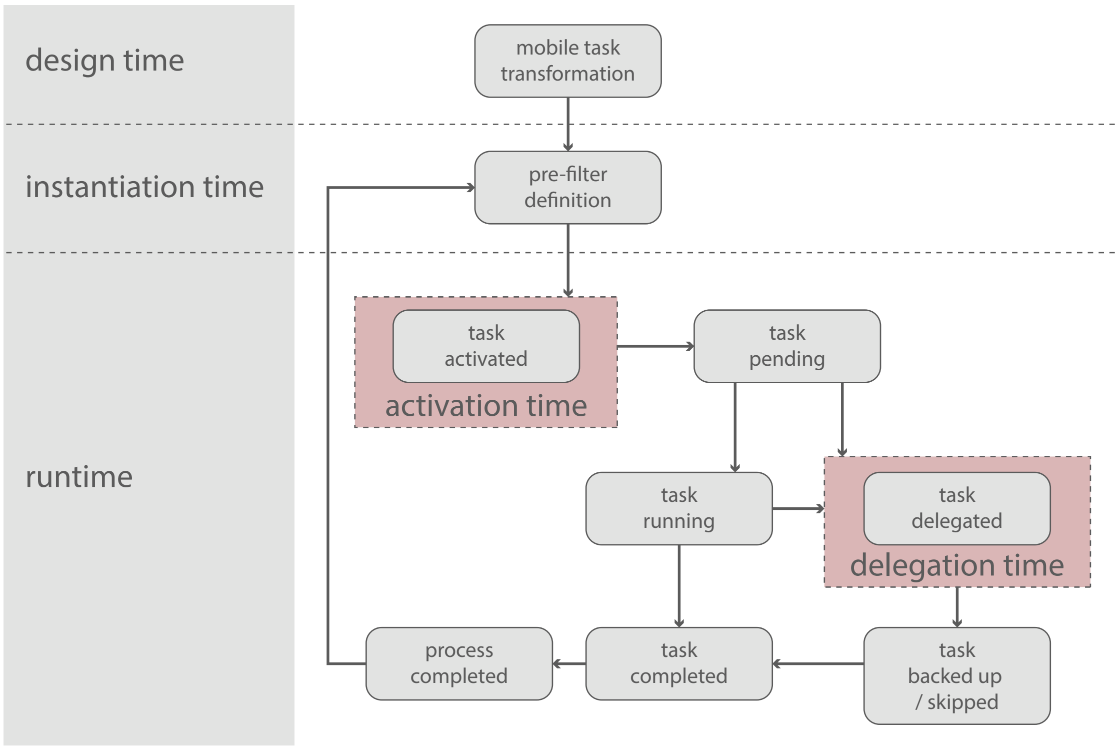


Figure 3.15.: Mobile task life cycle

The *MDS* introduced *activation time* and *delegation time*, two more specific time slots during run time in which the user lists are generated. Based on the mobile task life cycle, an architecture handling mobile task execution will be introduced in the next chapter.

4 A Generic Architecture for Mobile Task Execution

Today, process-based architectures are widely used in various scenarios. These systems are often intertwined with existing legacy environments and changes would imply high efforts regarding time and money. Approaches implying new implementations or changes in existing components, or the introduction of a completely new infrastructure are expensive to integrate into existing systems. For example, as soon as an interface has to be altered, it has to be verified against all other components in the system to avoid side effects. If a new environment has to be set up, all interfaces for legacy integration have to be changed and tested. A generic integration approach for mobile task execution, which provides interfaces for mobile clients as well as for existing BPMSs, is thus desirable. In this approach, current interfaces remain untouched to avoid interferences with other system components.

While related work often proposes new implementation approaches for mobile BPMSs, the need of integration paradigms for mobile task execution into existing process environments and live systems has not yet been discussed. This chapter will introduce a generic approach for the integration of mobile task execution. In Section 4.1 different integration and implementation concepts will be discussed, and the requirements for the target architecture will be identified in Section 4.2. Finally, Section 4.3 will explain all components contributing to the integration will in more detail.

4.1. Supporting Mobile Tasks in Business Process Management Systems

The foundation for all approaches providing mobile support in a process environment are the functionalities of contemporary BPMSs. The traditional architecture can be described as a centralized authority providing a *user repository*, *process definition tools*, and a *process engine* (cf. Figure 4.1 Ⓐ) [36]. All registered users and the according rights management are stored in the user repository.

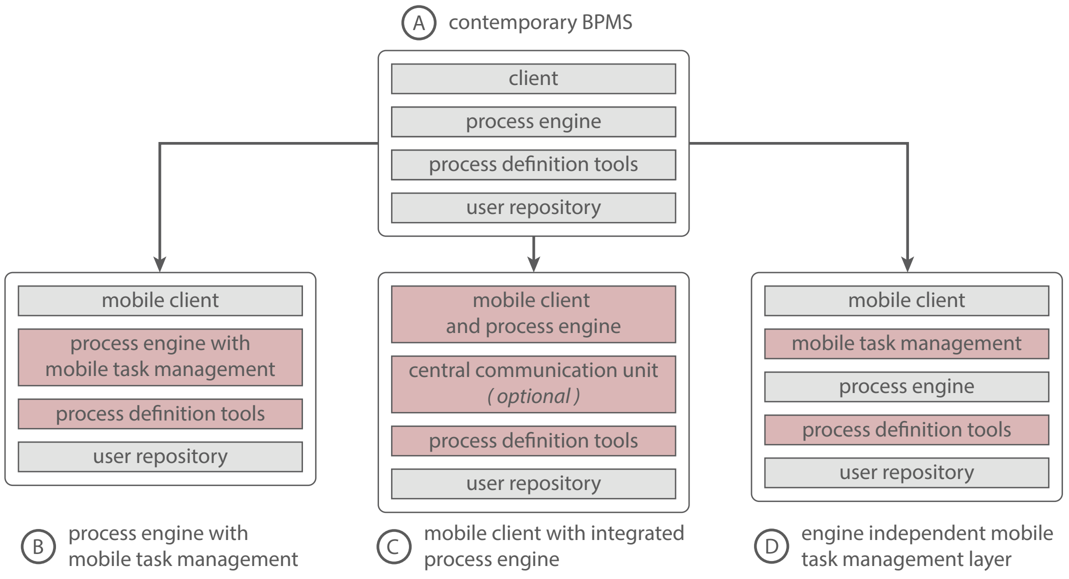


Figure 4.1.: Components of traditional and mobile BPMSs

The *process definition tools* are used to create process models at design time, which then can be instantiated by the process engine for run time. The process engine serves as an execution environment for process instances and holds the process repository, which contains all available process models. It provides interfaces to process initiation and execution control. Distributed, non-mobile clients enable users to initialize processes and perform tasks by using the engine's interfaces.

Whilst all base components can be found in related projects, the structure and implementation may differ crucially. So far, two approaches for introducing mobile execution support to a process environment can be identified. First, the reimplementation of a central process engine including a mobile task management as shown in Figure 4.1 Ⓑ (e.g., *Rome4U* [18]). Second, mobile clients running a process engine on each device and communicating either directly (e.g., in a MANET as done by *CiAN* [20] and *Sliver* [17]) or through a central communication unit (e.g., *MARPLE* [16] and *DEMAC* [15]) as shown in Figure 4.1 Ⓒ. In both cases, the process model has to be updated with additional functionalities for a proper representation of mobile tasks either by extending the existing concepts or by introducing a new notation. Thus, the process definition tools have to be updated as well.

The downside of such approaches is the lack of integration interfaces for legacy systems. All existing interfaces have to be changed and re-evaluated. The second approach, if not using a central communication unit, does not allow an integration at all, since it is designed for a certain infrastructure.

An engine independent approach is a promising way of tackling these shortcomings. Therefore, a new mobile task management layer between process engine and mobile clients is introduced (cf. Figure 4.1 Ⓓ). The engine delegates all upcoming mobile processes to the layer, which then handles the complete life cycle of this task at run time and manages the status of mobile devices and according users. Hence, a mobile client must communicate only with the management layer. For the communication between layer and process engine, the engine's existing communication interfaces have to be used (e.g., a HTTP interface) and therefore have to be implemented by the management layer. Following this approach, the only part of the basic BPMS which has to be altered, are the process definition tools. If the framework of Chapter 3 is used, only small efforts are necessary, since most of the used concepts can be implemented in existing business process languages.

4.2. Requirements

In the following sections, a concept for a mobile management layer which uses the mechanisms of Chapter 3 will be introduced. For this purpose, the requirements not just for the layer itself, but also for the used BPMS and the mobile clients, have to be addressed. The requirements for the management layer mostly emerge from the used framework and can be determined as follows:

L1 **Mobile user management:** The management layer has to provide a mobile user repository which has to be a subset of the repository of the underlying BPMS. Also needed is a generic authentication mechanism which evaluates authentication requests and forwards them to the underlying process engine.

L2 **Mobile device management:** All available mobile devices have to be managed by the mobile management layer. This includes a unique identification mechanism, a link between the device and the current mobile user who is using it, and tracking of the current device status (e.g., battery status or available sensors).

L3 **Handling mobile task life cycle:** The management layer has to support all run time states of the mobile task life cycle. Thus, the user list management, the delegation service, and the backup service have to be provided and it must also be possible to track the current state of a mobile task (e.g., provided data and status).

L4 **Process instance tracking (optional):** A desirable feature would be the possibility to track the status of a process instance with mobile tasks. This can then be used to evaluate the average user behavior at process instance level and, in addition to initiate global settings for all mobile tasks in a certain process instance (e.g., a global pre-filter).

Due to the integration context, only a minimal set of interfaces is required to communicate with the underlying process engine and the mobile clients:

IF1 User repository: The management layer must be able to access the underlying user repository and synchronize its own user repository.

IF2 Authentication: Authentication requests by a mobile client need to be accepted and can then be forwarded to the process engine. The engine's response has to be evaluated and provided to the client.

IF3 Process execution: All process and task related execution operations have to be provided. Task initiation requests by the process engine have to be accepted, task information has to be published to the mobile clients, and task status updates by the mobile clients have to be handled. The final execution results of a mobile task have to be provided to the process engine. Additionally, process instantiation information has to be gathered for process instance tracking.

IF4 Device status: The management layer needs access to a device's current status information (e.g., battery-status, location or sensor data), either by accessing this information directly or by letting the device publish it constantly or on request.

The underlying BPMS has to provide the components described in Section 4.1 and has to fulfill the following requirements:

W1 Process engine communication interface: The process engine has to provide a communication interface which can be used for authentication purposes and to initiate process models and to control business processes. The interface has to enable single process instances to communicate with external components (e.g., a REST service via HTTP).

W2 Accessible process language format: Mobile tasks can only be declared and the *mobile process transformation* and the automated dependency check performed if the definitions of the used modeling and execution languages (e.g., BPEL or BPMN 2.0) are known and accessible. Furthermore, either the process definition tools have to be extendible (e.g., with plug-ins) or the process model repository has to be accessible during design time.

Since mobile clients are used for presentation and interaction purposes only, the following user interfaces have to be offered:

- **UI1 Authentication:** A form where the user has to provide credentials in order to authenticate towards the process engine.
- **UI2 Working list:** A list containing all mobile tasks which can be performed by the current user.
- **UI3 Task execution:** A form which can be used to perform a mobile task by providing data and changing the task's state.

Additionally, the clients have to provide a communication interface which can be used to exchange data with the mobile management layer (e.g., via HTTP).

4.3. Integration Concept

Based on the requirements in Section 4.2, Figure 4.2 describes a concept to integrate mobile task execution into conventional BPMSs. A central aspect is the *Mobile Execution Layer (MEL)*, a service layer between the process engine and the mobile client. It handles the mobile task life cycle during run time, provides services for the mobile clients, implements interfaces for process engine related operations (i.e., authentication and user repository access), and provides task execution services which can be used by the process engine. Furthermore, the process definition tools have to be extended by a *mobile task transformation module*, which has access to the underlying process definition model and can be used to define mobile tasks as described in Chapter 3. The *mobile client* represents the user interface for mobile users. It provides a communication interface which is compatible to the *mobile client services* of the *MEL*. Contrary to the stationary client, which is provided by the BPMS, the *mobile client* communicates exclusively with the service layer. It does not implement any execution logic. Instead, all user interactions and provided data are sent to the *MEL* for further processing. A detailed view of the integration concept can be found in Appendix A, Figure A.1. Following this,

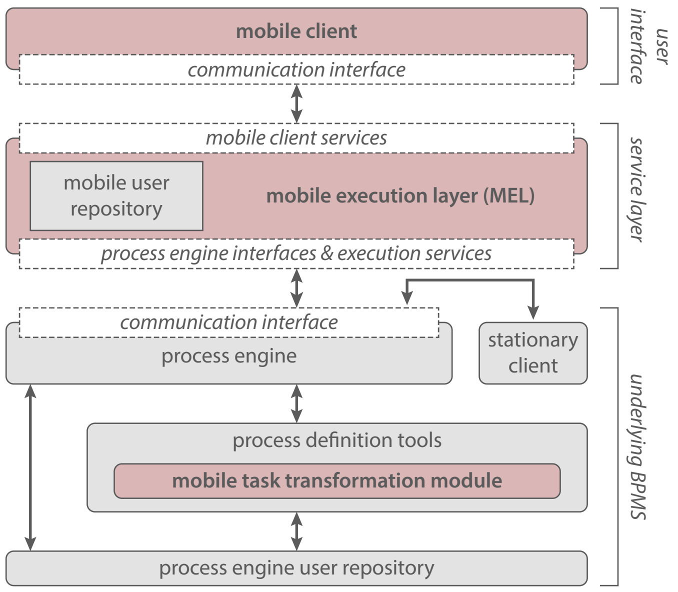


Figure 4.2.: Integration concept

the respective components will be discussed subsequently in more detail and the concept of *execution filters* will be introduced.

4.3.1. Execution Filters

Currently, during the life cycle of a mobile task, multiple concepts are used to define values and properties. On the one hand, every mobile task has properties such as a location or an urgency. On the other hand, pre-filters and operations are used during initiation time and delegation time. Tracking and evaluating these properties in a complex process structure can become more challenging than it should be, since all participating object classes in a mobile life cycle have to be taken into account, which are:

- The process instance
- The mobile task instance
- The mobile user list (ul_{mob}) and the mobile delegation list (dl_{mob})
- The backup operation / skip

To minimize the resulting complexity, the concept of *execution filters*, which can be assigned to each of the object classes, is now introduced. A filter is associated with one or multiple states in the mobile task life cycle and can only be applied there. Three types of *execution filters* can be identified:

1. **Process Instance Filters** can be used to define global default settings for all mobile tasks in accordant process instances. They can be overwritten by other filters.
2. **Task Instance Filters** can be used to define properties like urgency or the list threshold for a single mobile task instance. They can be used to overwrite *process instance filters*.
3. **User List Filters** can be used to manipulate the user list management of the *MDS* by adjusting the prioritization model and defining pre-filters.

With this set of filters, all concepts used in Chapter 3 can be represented. Implemented filters can easily be changed or the whole set can be extended without interfering with other components. Thus, a highly modular implementation becomes possible. All necessary execution filters fitting the requirements in Section 4.2 are listed in Tables 4.1, 4.2 and 4.3.

Process Instance Filters

Filter	**Mandatory**	**Description**
Default List Threshold	✓	Implements the *setThreshold()* method for mobile task transformation. *Can be overwritten by the corresponding task instance filter*.
Default Priority Weighting	✓	Defines the weights for priority calculation. *Can be overwritten by the corresponding task instance filter*.
Global Pre-Filter		Excludes a specific user for all mobile tasks in a process instance. The user will not be considered for user- and delegation list calculation.

Table 4.1.: Process instance filters

Task Instance Filters

Filter	Mandatory	Description
List Threshold		Implements the *setThreshold()* method for mobile task transformation.
Task Location		Implements the *setLocation()* method for mobile task transformation.
Urgency		Implements the *setUrgency()* method for mobile task transformation.
Skip	✓	Implements the *isSkippable()* method of the mobile task.
Force Skip		Forces a task to be skipped, even if a backup operation is necessary.
Offline Execution		Indicates whether a task can be executed as an *offline task*.

Table 4.2.: Task instance filters

User List Filters

Filter	Mandatory	Description
Location Matching	✓	Implements the nlf calculation.
User Behavior	✓	Implements the dbf and ebf calculation.
User Priority		Sets the upf for a specific mobile user.
User Constraints		Defines the cpf based on user constraints.
Priority Weighting		Defines the weights the priority calculation.
Local Pre-Filter		Excludes a specific user for a specific mobile task instance. The user will not be considered for the user- and delegation list calculation.
List Priority Calculation	✓	Combines other user list filters for user priority calculation ($Pr_{u_{mob}}$).

Table 4.3.: User list filters

So far, the execution filters can be defined globally on a process instance or locally on a single mobile task instance. To improve the flexibility, the current *execution status* of a process instance or task instances can be used as a precondition, if a filter is applied or not. An *execution status* is defined by:

- The status of data elements (e.g., the value of a data element)
- The current execution state of a task instance (i.e., pending, started, finished, backed up, skipped)
- An execution log, which holds records of all users, who worked on a task instance.
- Which filters were applied during execution.

A filter precondition can be denoted by boolean expressions based on these aspects. For example, a *local pre-filter* lpf on the mobile task $t1_{mob}$ will only be applied if task $t2$ has been finished and data element $d1$ has been written. Then the precondition for $t1_{mob}$ can be written as $pre_{t1_{mob}.lpf} = (t2.state \equiv finished) \wedge \neg(d1 \equiv null)$. lbf will be only applied if $pre_{t1_{mob}.lpf} \equiv true$ holds.

Execution filters and their preconditions can be defined during design time and instantiation time and will be evaluated and applied during run time. Hence, the *mobile task transformation module* and the *MEL* have to provide corresponding components for filter management.

4.3.2. The Process Engine and Process Definition Tools

Regarding the introduction of mobile task execution support by using a service layer, the underlying BPMS has to provide a minimal set of functionalities. This set is defined by the standard components of a BPMS [36] and the requirements **W1** and **W2** stated in Section 4.2. Furthermore, the process definition tools have to be extended by a *mobile task transformation* functionality. Figure 4.3 shows the structure of a BPMS which provides all necessary functionalities, and how its components interact with higher layers. Most modules are standard components and interfaces of a conventional BPMS. The execution environment and user management stay untouched, only the tools for the process definition have to be extended. Following this, all relevant components of a valid

BPMS and the internal and external communication patterns will be discussed in more detail.

Process engine user repository: The *user repository* holds the information of all registered users in the system. A user's identity, credentials and authorization information are non-transiently saved and can be accessed by other modules (e.g., by an authentication module). The repository does not have to differentiate between *stationary user* and *mobile user*. This will occur in a higher service layer.

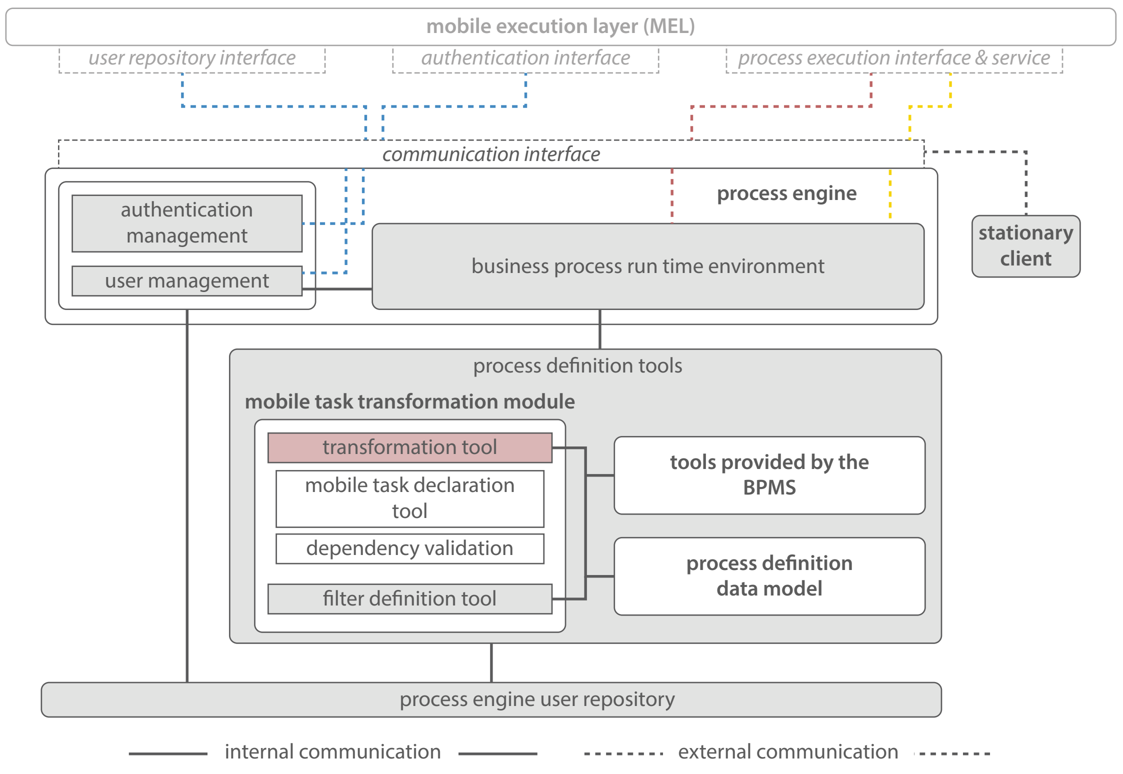

mobile execution layer (MEL)
user repository interface
authentication interface
process execution interface & service
communication interface
authentication management
user management
process engine
business process run time environment
stationary client
process definition tools
mobile task transformation module
transformation tool
mobile task declaration tool
dependency validation
filter definition tool
tools provided by the BPMS
process definition data model
process engine user repository
internal communication
external communication

Figure 4.3.: The underlying BPMS

BPMS process definition tools and data model: Process models based on a *process data model definition* (e.g., BPMN 2.0, BPEL) are designed by using the standard *process definition tools* provided by the BPMS. A finished process model is then transferred to the *business process run time environment* of the process engine, in which it can be instantiated and started.

Mobile task transformation module: This module extends the provided definition tools by enabling the process designer to declare mobile tasks, and run the *mobile task transformation* based on the underlying data model. Therefore, the *transformation tool* and the *filter definition tool* have to be implemented. The *transformation tool* is used to mark a task as *mobile* and provides an automated dependency validation in order to add a backup service. Execution filters can be declared and validated by using the *filter definition tool*. The module can be implemented as an integrated module or as an external application with access to the *process definition data model* and the process repository.

Process engine: The core module of a process engine is the *business process run time environment*. It manages the initiation and execution of processes and tasks, and manages a repository of all available process models. The engine provides a user management and authentication management, which inquire the *process engine user repository* to handle user authentication requests and task assignments based on the user rights management (e.g., Role Based Access Control (RBAC) [37]).

Communication interface: A communication interface has to be provided to enable external applications to access the user management and the process execution environment (e.g., to provide data or perform a task). It is used for the communication between the process engine and the *MEL*. It can also be used by user clients and external management tools.

Stationary client: A stationary client represents the standard user interface running on non-mobile devices. It enables users to authenticate themselves at the *BPMS* and to initiate and perform tasks. In particular, it uses the process engine's communication interface to interact with the execution environment and the user management. If a mobile task has to be backed up, it will always be delegated to a user on stationary clients.

4.3.3. The Mobile Execution Layer

The *Mobile Execution Layer (MEL)* is a crucial component of the integration concept. It is responsible for managing the *MDS* execution flow and therefore has to track the status of all available mobile devices and mobile users. It has to provide an execution filter management on both task and process instance level. Moreover, it acts as a communication relay between mobile clients and the underlying BPMS by providing services for both of them and implementing parts of the process engine communication interface. The *MEL* enables the tracking and analysis of mobile processes on a process instance level (cf. Figure 4.4). Henceforth, all components of the *MEL* shown in Figure 4.4 will be discussed subsequently.

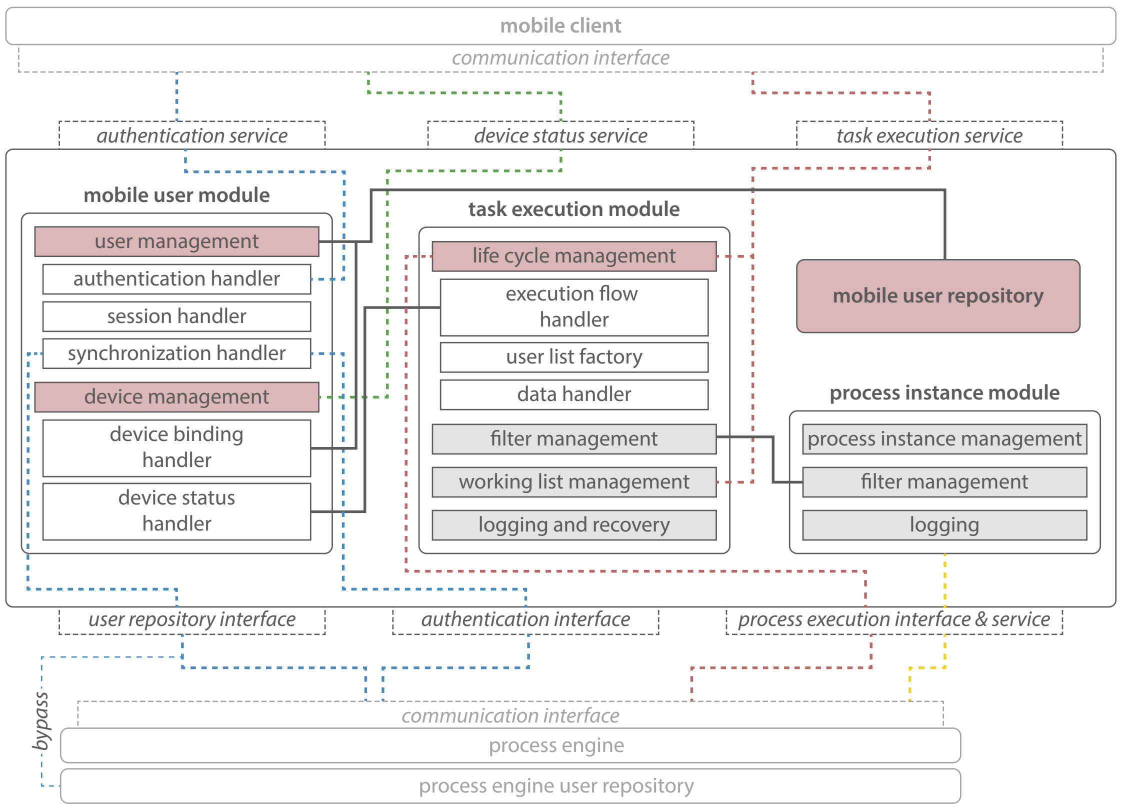


Figure 4.4.: The mobile execution layer

Mobile user repository: To avoid interferences with other components, the *process engine user repository* does not support the declaration of *mobile users*. Consequently, the *MEL* has to manage a *mobile user repository* which is a subset of the *process engine user repository* and references all users, who can be considered as *mobile users*. To reduce data redundancy, the repository holds only a unique identifier for each mobile user, which has to be provided by the *process engine user repository*. All additional information is gathered during the authentication process. Furthermore, the repository persists device bindings, which are used to identify authorized mobile devices. Thus, a unique identifier (e.g., a device's MAC address) and a shared secret (e.g., a security token) are saved for each authorized mobile device.

Mobile user module: The *mobile user module* provides *user management* and *device management* functionalities. The main task of the *user management* is to implement a model for mobile users, to manage online users, and to bind online users to devices. The *authentication handler* administers the authentication process by handing over a user's authentication requests to the corresponding module of the *process engine*. If the request was verified by the *process engine*, the *session handler* initiates a session for this user which holds the user's credentials and a temporary shared secret (e.g., a token). This enables the *MEL* to impersonate the mobile user towards the user management of the *process engine* for future communication (cf. Figure 4.5).

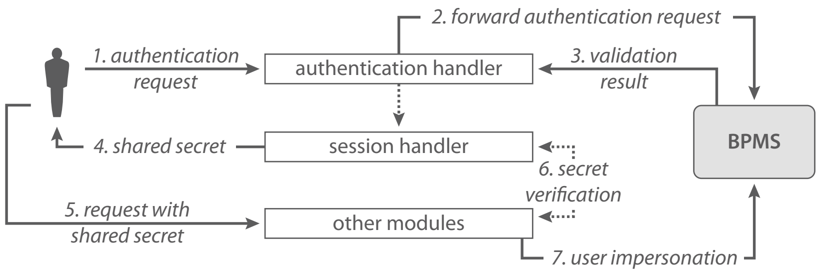


Figure 4.5.: User authentication and impersonation

The synchronization of the *mobile user repository* with the *process engine user repository* is conducted by the *synchronization handler*. For this purpose, operations are provided to request all user data from the BPMS and to mark certain users as *mobile users*.

To be able to control access not just on a user level, but also on a device level, all participating devices have to be bound to the system. Thus, a device has to send a *binding request* containing a unique identifier (e.g, MAC address). The request is then validated by the *device binding handler* of the *device management*. In the case of a positive validation, namely if the device was authorized by an administrator, a permanent shared secret will be exchanged, which has to be provided by the device in all future communication attempts. If one side looses the shared secret, the device has to send a new binding request, to re-initiate the binding process.

A crucial aspect during a mobile task execution is the status of mobile users, which is used for user list calculation. In most cases, a mobile user's status can be derived from the status of the device he is using (e.g., connectivity or location). The *device status handler* thus tracks all participating devices by gathering information about their connectivity, location, battery health and available sensors which can then be accessed by other modules (e.g., the life cycle management). Different approaches are possible for collecting the device status information:

- The status is requested by the *MEL* as soon as it is needed.
- Once a mobile user is authenticated, the *MEL* requests the device's status constantly.
- A mobile device has to send *alive messages* at a specific interval containing the devices current status information.

Using the first approach, an additional mechanism to detect connection losses is needed. The other two approaches imply this, since a constant polling mechanism is used. In particular, the middleware sided polling uses a request-response mechanism and therefore increases the complexity and generates a higher network load. As opposed to this, the client alive messages use a single message communication which nominates this approach as the most desirable method to track the status of mobile devices.

User repository interface: To be able to access the *process engine user repository* for user synchronization, *MEL* has to implement parts of the process engine's communication interface. If the communication interface does not provide access to the user repository, a *bypass* has to be implemented, for instance, a direct access to the persistence layer (cf. Figure 4.4).

Authentication interface: To be able to forward authentication requests and impersonate a mobile user, the *authentication interface* implements the user management modules of the process engine's communication interface which can then be used by the *mobile user module*.

Task execution module: The mobile task execution is mainly administered by the *task execution module*. It consists of four sub modules, handling different aspects of the execution process (cf. Figure 4.4). The *life cycle management* represents the *MDS*. It implements an *execution flow handler*, which controls the *MDS* execution flow and verifies state changes. It triggers automated state transitions by listening to the device status changes provided by the *device status handler*. The *user list factory* calculates the mobile user- and the mobile delegation lists. The data fields of mobile tasks are managed by the *data handler*. When a mobile user provides data for a task, it will be cached by the *data handler* for further processing. Thus, if a task is delegated, the previously entered fields can be provided to the new user. The data will not be published to the process engine until the task has been finished or has to be backed up.

When a mobile task is instantiated, the *filter management* will evaluate all attached execution filters. Furthermore, it applies *process instance filters*, which are used to set default values, and all *task instance filters*. During execution, the *user list factory* consults the *filter management* to gather the *user list filters* in order to calculate and prioritize the user lists.

All mobile tasks which can be performed by a particular mobile user have to be published in a way in which they can be efficiently processed by a mobile client. Therefore, the *working list management* organizes them in working lists, with one list each per mobile user. A list entry reveals information about the identifier (e.g., a unique task ID), the name, the data dependencies, and the current status of a mobile task. The entries are distinct, which means, that the same task can be only once in the same working list. However, one task can appear in multiple distinct lists. The working lists are linked to the *MDS* execution flow and will be updated each time a task state transition occurs, or the mobile user list or the mobile delegation list is recalculated.

Finally, the *logging and recovery module* implements the recovery strategy for handling system crashes. Hence, log files have to track all active tasks which have not yet been published to the BPMS. To recover all properties of a running task, log entries for state transitions and provided data have to be considered.

Process instance module: With regard to tracking the mobile execution on a process instance level, processes can be registered in the *process instance module* once they are instantiated. On registration, all *process instance execution filters* have to be advertised and additional information (e.g., total number of mobile tasks) can be provided. This information can then be linked to the task execution to enable overarching tracking and analyzing functionalities.

Process execution interface & service: The communication between the *MEL* modules and the BPMS is based on the implementation of parts of the process engine's *communication interface* and on services which can be used by the process engine. The *process execution interface* implements the components of the *communication interface* used to preform single tasks. It has to be able to finish a started task and write all depending data fields. For process and task instantiation, the *process execution service* is provided. It has to use a transfer protocol which is supported by the *communication interface* (e.g., HTTP). The service accepts instantiation requests which will then be forwarded to the *task execution module* or the *process instance module* (cf. Figure 4.4). While a process instantiation request has to provide only a unique process identifier and a list of execution filters, a proper task instantiation request has to provide the following information:

- a unique task identifier
- the task's name
- all data dependencies
- a list of execution filters
- the initial user list
- the unique identifier of its parent process instance

To track the whole process instance life cycle, a *process finish notification* has to be sent, once a process instance has reached its final state. However, communication between the process engine and the *MEL* only occurs during the execution of a mobile process:

1. when a process is instantiated,
2. if a single mobile task is started,
3. if a mobile task is finished or has to be backed up, or
4. when a process instance finishes.

Client services: Mobile clients can use the client services, i.e., the *authentication service*, the *device status service* and the *task execution service* to communicate and interact with the *MEL*. Hence, authentication requests have to be sent to the *authentication service* that forwards them to the user management. The *device status service* accepts *alive messages* from the devices and forwards them to the device management, while a mobile client may publish task progress and data values to the *MEL* via the *task execution service*. Moreover, the service is used to distribute the working lists to the mobile clients.

4.3.4. The Mobile Client

Contrary to other integration approaches, where the mobile client is an essential part of the execution processing (e.g., MARPLE [16]), the integration concept described in this chapter uses mobile clients only as a user interface. However, to fulfill the requirements **UI1** to **UI3** in Section 4.2, a well-defined communication and data processing between the mobile client and the *MEL* is mandatory.

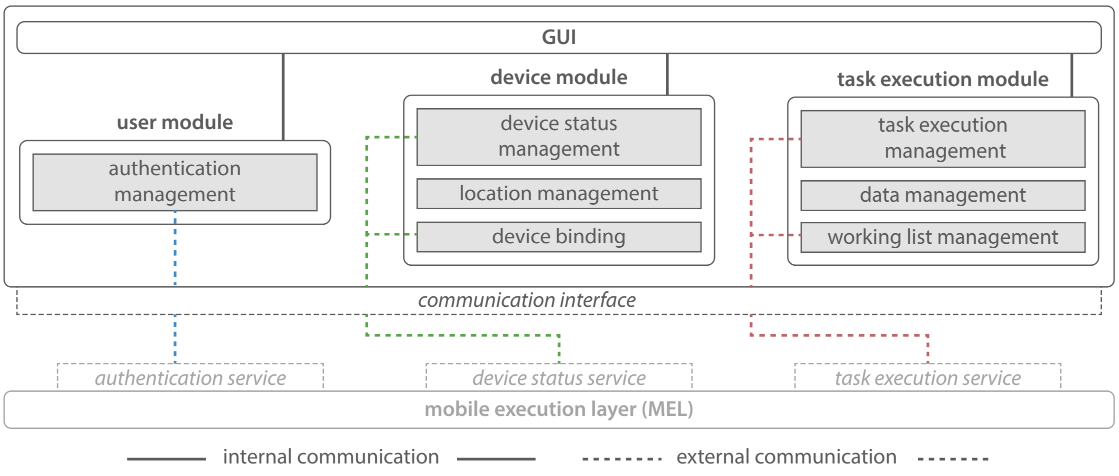


Figure 4.6.: The mobile client

Figure 4.6 shows a three-layer model of a mobile client based on the Model View Controller (MVC) pattern. It consists of a Graphical User Interface (GUI), three internal management modules and a communication interface. The *GUI* enables entering user credentials in order to authenticate to the system, to view a user's working list, and task details as well as to enter data in order to perform a task. All visible information is gathered, and user input is processed by the underlying management modules.

The *user module* handles the authentication process by taking the user input from the *GUI* and compiling it into a message format which is accepted by the *authentication service*. Once a user is authenticated, the module provides all information needed for further communication attempts.

The main task of the *device module* is to track the current device status and location, and provide this data to the *MEL* by sending *alive messages*. Additionally, it invokes device binding requests and holds the device binding information, i.e., the shared secret and the MAC address.

To perform a mobile task, the *task execution module* requests the working list and further task information from the *MEL*. The data, which is provided by the user, will be verified and forwarded by the *data management*. Furthermore, the *task execution management* handles task state transitions, namely if a user wants to start or finish a task.

Finally, the *communication interface* is used to interconnect the client with the *MEL*. It has to implement the same transfer protocol as the client services (e.g., HTTP) and is used by the management modules as a single access point.

In this chapter, a concept for the integration of mobile task support into BPMSs and the requirements for such integration have been introduced step by step. The full structure can be reviewed in Figure A.1 in Appendix A. To show the feasibility of this concept, it was partially implemented as a prototype, which will be discussed in the following chapter.

5 Proof of Concept

The integration concept, proposed in the last chapter, aims to have as little impact as possible on the underlying BPMS by managing the execution of mobile tasks in an intermediate service layer. A *MEL* prototype focusing on the execution time of the mobile task life cycle has been implemented to show the feasibility of this approach. It uses the REST framework *play* [38] as execution environment and service provider, and the java-python integration library *jython* [39] for the implementation of a freely programmable engine interface. A GUI- and an interaction concept for mobile clients have been developed to show possible interaction patterns and how information provided by the *MEL* could be presented.

This chapter addresses the implementation details of the prototype's core features, its shortcomings and the interaction concept. To show the significance of the impact on the underlying BPMS, multiple integration scenarios will be discussed regarding provided interfaces and supported communication patterns. Finally, these scenarios will be evaluated by applying them to three actual BPMSs: *Activiti* [40], *JBoss jBPM6* [41] and *AristaFlow* [42].

5.1. MEL Prototype

The REST framework *play* [38] was used for the implementation of the *MEL* prototype. It provides a lightweight application server for applications written in java or scala [43], a complete REST API supporting all standard HTTP actions (i.e., *GET*, *POST*, *PUT* and *DELETE*), a session management, and object relational mappers (i.e., *EBean ORM* and *JPA*). Regarding the limited resources of mobile devices, using a lightweight communication model such as REST is a promising approach. However, since REST

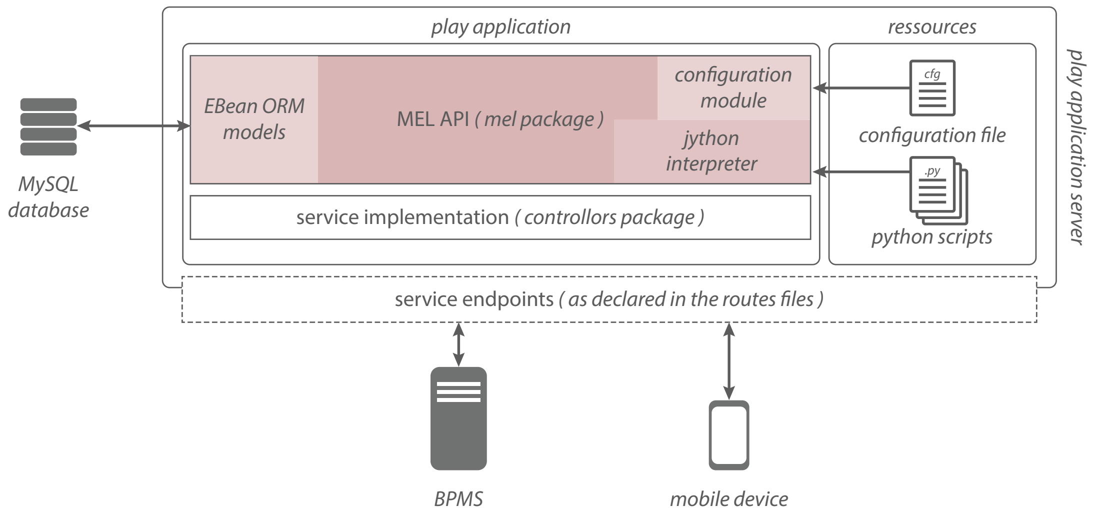


Figure 5.1.: The *MEL* prototype deployed on a *play* application server

paradigms are not supported by all legacy or established BPMSs, a freely programmable data translation interface based on python scripts is provided. The java-python integration *jython* [39] is used to integrate the scripts into a java environment. Figure 5.1 illustrates the structure of the deployed *MEL* application. The core functionalities of *mobile user management*, *mobile device management* and *task execution management* are located in the *MEL API*. It also manages the required EBean ORM models and initiates the *jython* interpreter. The configuration module loads global configurations, and provides methods to edit them (all available configuration options can be found in Appendix C). A configuration user management is in place to manage access to the configuration. Python scripts and configuration files are placed in a public resource folder which can be accessed by the *MEL API* and will be loaded on demand. To define endpoints for services, *play* provides the *routes file*. The implementations of all services listed in this file can be found in the *controllers* package. They use the *MEL API* and the *jython* interface to process requests and return valid responses to the underlying BPMS and mobile clients. The features supported by the prototype are listed in Tables 5.1 and 5.2. Core features will subsequently be discussed in more detail.

Mobile Users	
Authentication and impersonation	✓
Session-handling	✓
User repository synchronization	✓
Mobile Devices	
Device binding management	✓
Device status tracking	✓
Task Execution	
Execution flow handling	✓
User list generation	✓
User list prioritization model	✓
Data handling	✓
Execution filter management	✓
Execution filter preconditions	×
Offline tasks	✓
Working list management	✓
Logging and recovery	×

Table 5.1.: Supported prototype features (1)

Process Instance	
Process instance management	×
Process instance tracking	×
Logging	×

Table 5.2.: Supported prototype features (2)

5.1.1. REST Services

As mentioned before, the communication interfaces for the mobile clients and the underlying BPMS are implemented as REST services. *JSON* is used as data format to keep the communication footprint on a minimum and benefit from the advantages of the *jackson JSON parser*. The client services, described in Tables 5.3 and 5.4, provide fundamental functionalities for authentication, device status tracking and task execution on mobile clients. Before being able to use other client services, mobile users have to be

Authentication Services

Endpoint	Method	
`/auth/login`	POST	Accepts a mobile user's credentials for authentication. If the authentication was successful, a user session, which will be used for further identification, is initiated.
`/auth/logout`	GET	Removes the current user session.
`/auth/dam`	POST	A device has to send DAMs[1] constantly, containing the mac-address, the current location and supported sensors.

Table 5.3.: Services for mobile clients (1)

authenticated by the *MEL* to initiate a user session by using the according *authentication service*. Although *play* is a stateless middle-ware, it provides a session handling by adding an encrypted cookie to each response, which then has to be included in every following request sent by a client [38]. The authentication information is stored in the

[1] Device Alive Message (DAM)

session and will be used for all further service calls. As soon as a user is authenticated, the device has to continuously send Device Alive Messages (DAMs) to the `/auth/dam` endpoint within a certain timeout. This will ensure that the mobile user is considered to be *online*.

Task Execution Services

Endpoint	Method	
`/task/<taskID>`	GET	Returns detailed information about properties, execution status and data fields of the requested mobile task.
`/task/<taskID>`	POST	Changes the task's execution state to *STARTED* and sets the current session user as person responsible. The mobile task will be removed from all working lists except the one of the current session user.
`/task/<taskID>`	PUT	Accepts data field updates by the responsible user. The data will not be populated to the BPMS yet, but cached by the *MEL* and provided on further requests.
`/task/<taskID>`	DELETE	Changes the task's execution state to *FINISHED*, populates all data fields to the BPMS and removes the mobile task from all working lists.
`/working list`	GET	Returns the working list for the current session user.

Table 5.4.: Services for mobile clients (2)

After a successful authentication, a mobile client can use the services listed in Table 5.4 to get the working list of the currently logged in user, to start and finish a mobile task and to write data fields. A mobile task can be identified by its unique task ID, which has to be provided as resource identifier in the respective service endpoints, noted as `<taskID>`.

To maintain a better compatibility with older BPMSs, which do not support the REST paradigm, the BPMS services in Table 5.5 use only the more common HTTP methods *GET* as well as *POST*, and accepts plain text data. The *POST* service initiates a mobile task which will then be available to the mobile clients for execution. Before initiating a

BPMS Services

Endpoint	Method	
`/engine/task/<taskID>`	POST	Accepts mobile task initiation requests, containing the task's unique ID, name, description, a set of data fields, a initial set of users and a set of execution filters. On a valid request, a new mobile task will be initiated and activated.
`/engine/task/<taskID>`	GET	Returns all information about the requested mobile task, if task polling is activated and the task has already been finished.

Table 5.5.: Services for the underlying BPMS

task, the request sent by the BPMS will be parsed by the *Freely Programmable Interface Layer (FPIL)*. The communication data format can thus be chosen freely by the BPMS, which increases legacy compatibility even more.

The prototype supports two methods for publishing a finished mobile task to the BPMS: First, the task is handled by the *FPIL* and therefore will be sent directly to the communication interface of the BPMS. Second, all finished tasks are provided by a service, which can constantly be polled by the BPMS (cf. Table 5.5, *GET*). To reduce redundancy and to avoid side effects, only one method, which can be set in the *MEL* configuration file, can be used at the same time.

BPMS Services

Endpoint	Method	
`/config/mobileusers/import`	GET	Returns a list of all available BPMS users.
`/config/mobileusers/<userID>`	GET	Returns information about the requested mobile user.
`/config/mobileusers`	POST	Adds a BPMS user to the mobile user repository.
`/config/mobileusers/<userID>`	DELETE	Removes a mobile user from the mobile user repository.

Table 5.6.: Services for the synchronization of the mobile user repository

In addition to the previous services, the *MEL* provides a configuration service interface for the configuration module. It is only accessible for configuration users which represent a privileged user role within the *MEL*. The interface provides an authentication method similar to the one used by the mobile client services. After authentication, a configuration user has access to services for managing configuration users and mobile device bindings and for synchronizing the engine user repository with the mobile user repository (cf. Table 5.6). A full list of all available services can be found in Appendix B.

5.1.2. Freely Programmable Interface Layer

For decoupling the *MEL* from the underlying BPMSs, the prototype provides a *Freely Programmable Interface Layer (FPIL)*. It manages data translation interfaces that are either used by services to translate incoming data into a format which can be handled by other modules (i.e., data transformation interfaces, cf. Figure 5.2 Ⓐ), or by modules to send data with a supported data format directly to the underlying BPMS (i.e., communication interfaces, cf. Figure 5.2 Ⓑ). The interfaces are implemented as python scripts and loaded on demand from a public resource folder. The java-python integration *jython* is used for the execution within the *play* java environment. It maps the scripts to

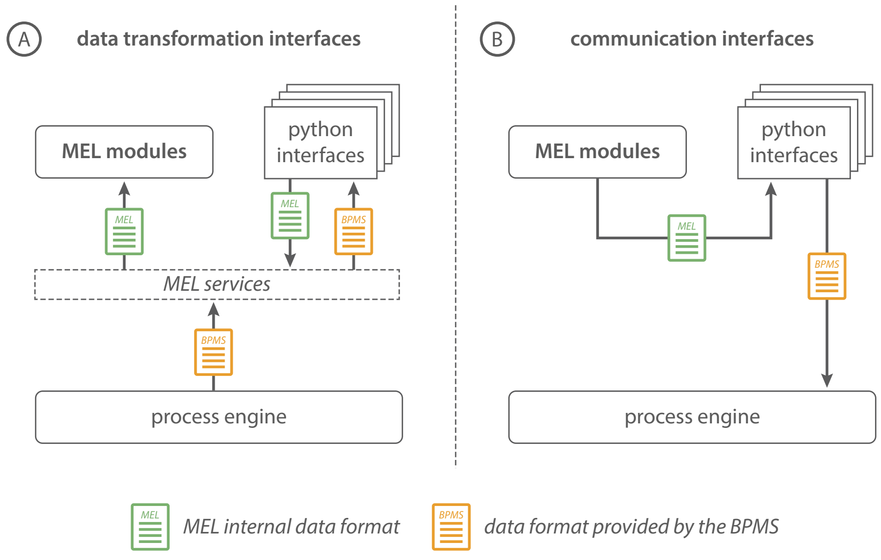


Figure 5.2.: Data translation interfaces

java interface classes to load and run python scripts and to use the results within a java implementation. Hence, all python scripts have to implement one of the supported java interfaces provided by the `mel.common.interfaces.engine` package:

- **IAuth**: Used by the authentication service to pass user credentials to the BPMS. Scripts have to implement `authenticate(username, credentials)` which accepts the user name as string and further credentials as an array of strings. On success, the method has to return the user's engine ID or -1 on failure.
- **ITaskActivation**: Used by the task activation service to translate incoming data into the json format. Scripts have to implement `toJson(body)`, which accepts the HTTP request body as string and returns a valid json string.
- **ITaskExecution**: This interface is used, if polling of finished tasks is deactivated. Scripts have to implement `callRPC(taskId, finalStatus, Data, userEngineId, userName, credentials)` with the task data and the user credentials as parameters, which will then be sent to the communication interface of the BPMS.
- **IUserManagement**: Provides the `getUser()` method which queries all users in the user repository of the BPMS. This interface is used to synchronize the mobile user repository.

For each interface, multiple implementation scripts can be placed in the according folder. The script to be used is set in the *MEL* configuration file. This enables the prototype to support multiple BPMSs without changing their implementation details. In addition to the java interface mapping, *jython* provides a full integration of java into python. This enables the freely programmable interfaces to use java APIs within python scripts [44].

5.1.3. Life Cycle Management

The main focus of the *MEL* prototype is on the *life cycle management*. This includes initiating mobile tasks and managing their execution flow, calculating the mobile user lists and managing the working lists for all active mobile users. For this reason, the *MEL*

API provides multiple manager classes and implements mobile tasks as parallel running threads. Figure 5.3 illustrates the initiation process of a mobile task. The *task execution manager* provides a method which is used by the *activation service* to initiate a new mobile task. The *mobile task* class implements the `java.lang.Runnable` interface and manages its own execution state, data dependencies and execution filters. To initiate a task, all requested task instance filters will be applied and the thread will be started and added to a managing list within the *task execution manager* containing all active mobile tasks. When started, the *mobile task* calculates the mobile user list ul_{mob}, sends an update notification to the *working list manager* and changes its execution state to *PENDING*. It then waits until the task has been finished by a user or until the task is due. It will only be activated for device status updates and data field updates. The *working list manager* is responsible for all working lists of active mobile users and provides them through the *working list service*. When notified by a new mobile task, the manager will add this task to all working lists of active mobile users in ul_{mob}.

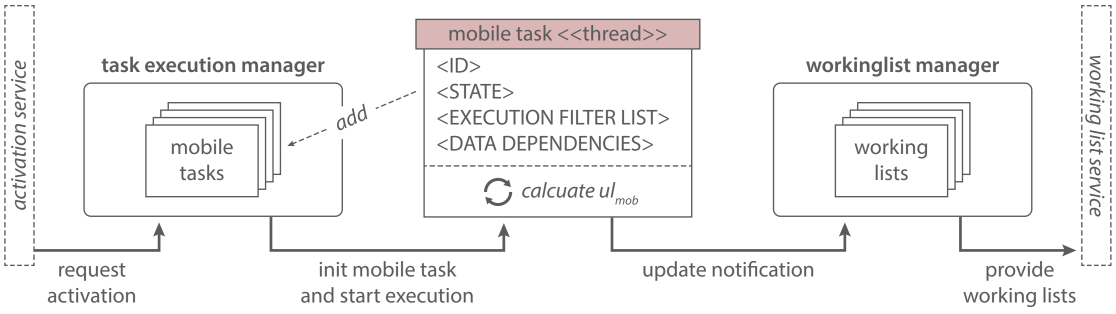


Figure 5.3.: Mobile task initiation

After task initiation, the further task execution is handled as shown in Figure 5.4. As soon as a mobile user starts to work on a task t, the *task execution service* requests a state change of t to *STARTED*. The request will then be forwarded to the mobile task thread. If a task has already been started or delegated, the *task execution manager* has to listen for mobile device state changes of performing users.

A mobile device is considered to be in one of four states: *ONLINE, PENDING, OFFLINE* and *UNBOUND*. A device with no active user session (i.e., no user is logged in on this device) is *UNBOUND*. As mentioned before, device status tracking is done by collecting Device Alive Messages (DAMs) of all devices which are not unbound. DAMs are managed by the *device activity manager* and have a *device alive listener* attached. There is only a single DAM for each device d, which will be refreshed as soon as d sends a new message. If the manager receives an initial DAM (i.e., when a user has logged in), the according *device alive listener* will notify the *task execution manager*. The latter will then publish these changes to all pending mobile tasks so that they can refresh ul_{mob} if the new user is in ul_{init}. After handling an initial DAM, the respective device will be considered as *ONLINE*. If d fails to send the next DAM within a configurable *alive timeout*, this device will be considered as *PENDING*. This status was introduced to cope with short time shutdowns or connection losses and reduces the amount of unnecessary delegations. Technically, the device is offline but the delegation will be postponed to give the mobile user the chance to reconnect to the network within a *pending timeout*. If this fails, the device will be considered as being *OFFLINE* and offline time measurement will be started. Additionally, the *device alive listener* notifies the *task execution manager*

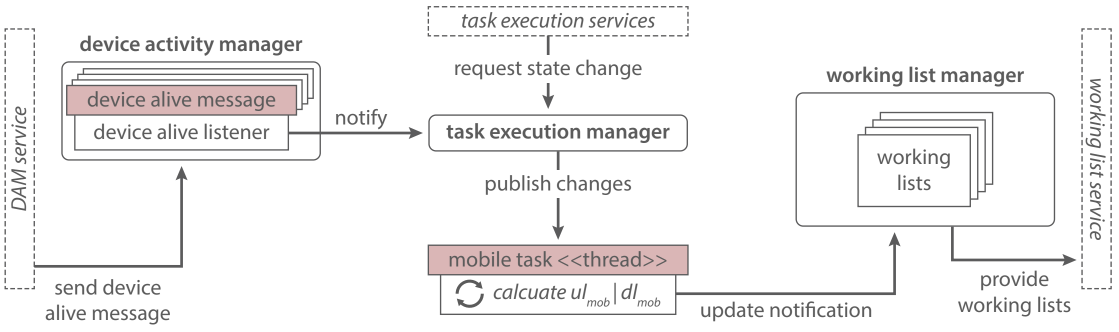


Figure 5.4.: Handling mobile task execution

which will then publish the changes to all pending tasks and all started or delegated tasks currently performed by the mobile user who just went offline. While pending tasks will recalculate ul_{mob}, started and delegated tasks, unless they are offline tasks, will calculate dl_{mob} and set the best fitting user in charge. Following the list calculation, the *working list manager* is notified to update all affected working lists. If the offline device fails to reconnect after an *offline timeout*, it will be considered as *UNBOUND* and the respective user session will be terminated. Otherwise, the device status is *ONLINE* and it can be used for further user list calculations. In both cases, the offline time measurement will be stopped and the result will be saved and may be used for following list prioritization runs.

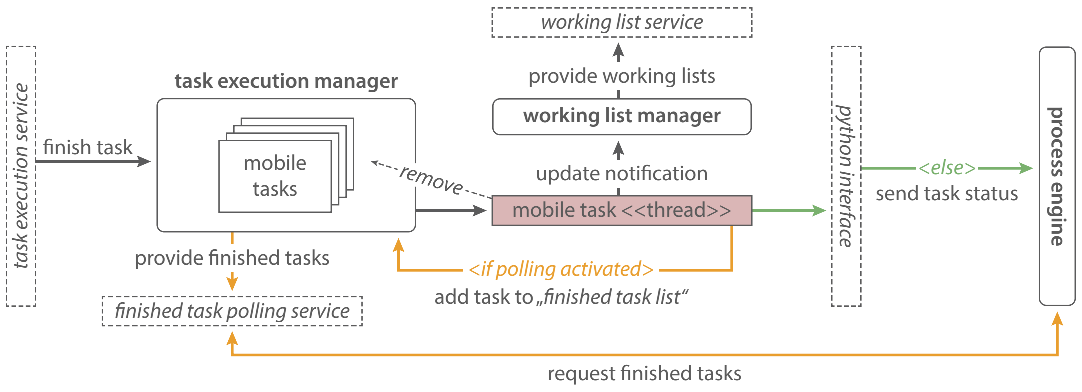


Figure 5.5.: Finishing a mobile task

If a mobile task has to be finished, the *task execution manager* removes the respective thread from the managing list and notifies the task to initiate the finishing procedure (cf. Figure 5.5). The mobile task therefore sends an update notification to the *working list manager*, which will remove this task from all working lists. If task polling is activated, the mobile task instance will be added to the *finished task list* of the *task execution manager* and can be queried with the *finished task polling service*. Otherwise, the task data will be sent directly to the BPMS using the configured python interface and the thread execution will be terminated. This procedure can be invoked manually by a mobile client, using the *task execution service*, or automatically by the mobile task instance, if a backup is required.

5.1.4. Mobile Clients

The integration concept in Chapter 4 has introduced mobile clients as a user interface on mobile devices, such as mobile phones or tablets. A GUI prototype has been developed to show possible interactions with the *MEL* prototype regarding the user interface requirements listed in chapter 4.2. Figure 5.6 shows the corresponding interaction model as a flow chart, where nodes represent different user views, and transitions user interactions. The *login* view is the application's entry point. It shows a simple form for entering

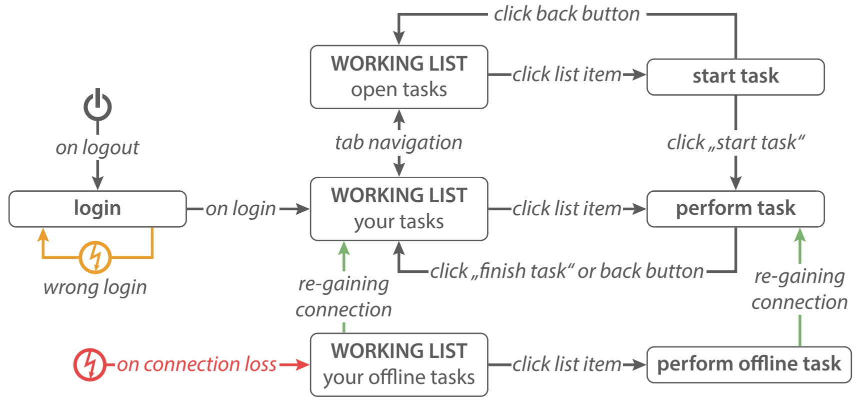
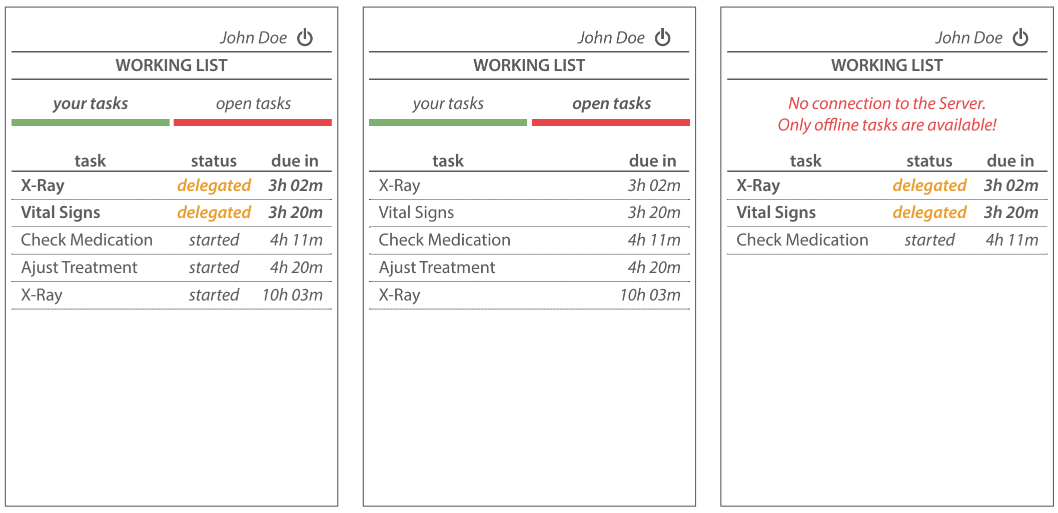


Figure 5.6.: Interaction model for the mobile client

user credentials. This form has to be adapted to the needs of the underlying BPMS. The BPMS *AristaFlow* [45], for example, needs a user's name, role and password for authentication, which has to be reflected by the login form.

After a successful login procedure, the *working list* view will be entered, showing the current user's working list. It is divided into two sub-views to improve visibility and to reduce the cognitive load of the user. One shows started or delegated mobile tasks the user is working on (cf. Figure 5.7a), and another activated tasks which can be started by the user (cf. Figure 5.7b).


John Doe ⏻

WORKING LIST

your tasks *open tasks*

task	status	due in
X-Ray	***delegated***	***3h 02m***
Vital Signs	***delegated***	***3h 20m***
Check Medication	*started*	*4h 11m*
Ajust Treatment	*started*	*4h 20m*
X-Ray	*started*	*10h 03m*

John Doe ⏻

WORKING LIST

your tasks ***open tasks***

task	due in
X-Ray	*3h 02m*
Vital Signs	*3h 20m*
Check Medication	*4h 11m*
Ajust Treatment	*4h 20m*
X-Ray	*10h 03m*

John Doe ⏻

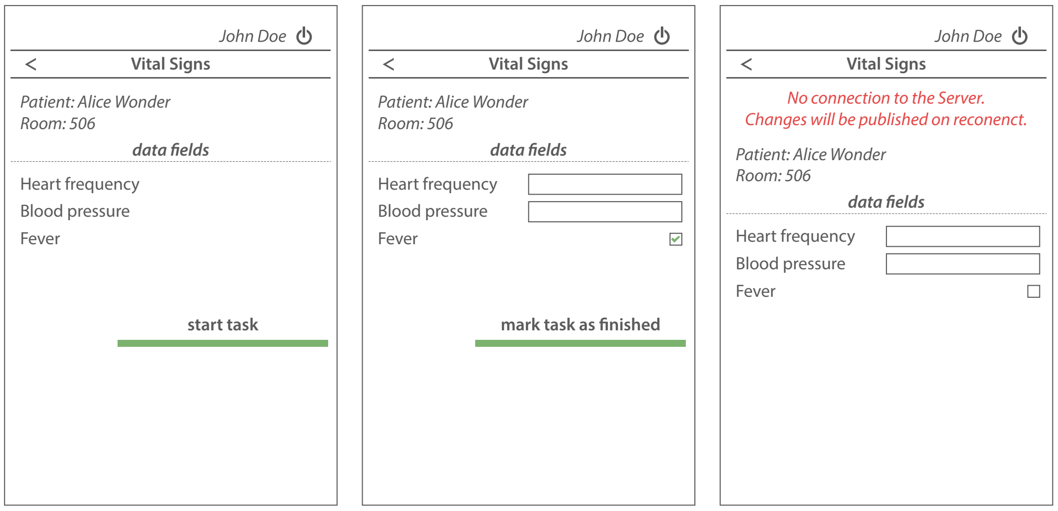
WORKING LIST

No connection to the Server.
Only offline tasks are available!

task	status	due in
X-Ray	***delegated***	***3h 02m***
Vital Signs	***delegated***	***3h 20m***
Check Medication	*started*	*4h 11m*

(a) Your tasks (b) Open tasks (c) No connection

Figure 5.7.: Working list views



No connection to the Server.
Changes will be published on reconenct.

(a) Start a task (b) Standard task view (c) Offline task view

Figure 5.8.: Task views

A tabbed navigation bar has been used to switch between the two sub-views. The initial list is sorted by due time to indicate the urgency of tasks. Touching the table headers will sort the list by name or status. A third working list view is displayed if devices loose the network connection (cf. Figure 5.7c). In this state, only started offline tasks are available to the user. This error view will be entered on any connection loss, regardless of view displayed before.

By touching a working list entry, a task view will be entered. If a pending task is selected, the view will display fundamental task information such as task name, description and needed data fields (cf. Figure 5.8a). When pressing the *start task* button, the client will send a status change request to the MEL. After a task has been started, the view has to provide possibilities to enter required data fields and to mark the task as finished (cf. Figure 5.8b). Suitable input fields have to be displayed to handle different data types (e.g., text fields for string values, checkboxes for Boolean values). Entered data will be transferred to the *MEL* immediately. If the task view is entered while the device is not connected to the network, entered data will be cached and sent to the *MEL* as soon as it reconnects. It is not possible to mark a task as finished, as long as the device is not connected (cf. Figure 5.8c). The task view can be left at any time by pressing the back button next to the task name. This, as well as marking a task as finished, will bring the user back to the working list view. Pressing the logout button next to the user name in the top right corner will log the user out of the system by removing the *MEL* user session and bring her back to the initial login view.

5.2. Integration Scenarios

To evaluate the prototype's generic approach and the feasibility of the integration concept, they have to be applied to actual BPMSs. In this context, various integration scenarios can be identified emerging from differing communication interfaces and organization models of BPMSs. Since the pitch points of the *MEL* and the underlying BPMS are well defined, the integration scenarios can be classified by the following aspects:

User repository access (URA): The *MEL* needs access to the user repository of the BPMS to synchronize the mobile user repository. This can be done either through interfaces provided by the BPMS (e.g., a SOAP web service) and implemented as part of the FPIL, or by using a direct access to the persistence layer of the organization model, bypassing the access control of the BPMS. The latter will increase the complexity and error-proneness of the user management interface, since the detailed data structure of the used organization model has to be known and handled.

Authentication ($AUTH$): BPMSs have to provide an authentication interface which the *MEL* can use to authenticate and impersonate a mobile user. There are no restrictions in the used authentication method, since an arbitrary number of generic user credentials is supported. Depending on the capabilities of the mobile devices and the implementation of the authentication interface used by the FPIL, even a multi factor authentication (e.g., username, password and swipe card) is feasible. The absence of an authentication interface is unlikely, taking into account that most BPMSs are designed for distributed environments.

Supported communication patterns and protocols (COM): While the data format used for the communication between process engine and *MEL* will be translated by the FPIL and can thus be chosen freely, all underlying BPMSs have to support basic HTTP features to communicate with the services provided by the *MEL*. Regarding the initiation

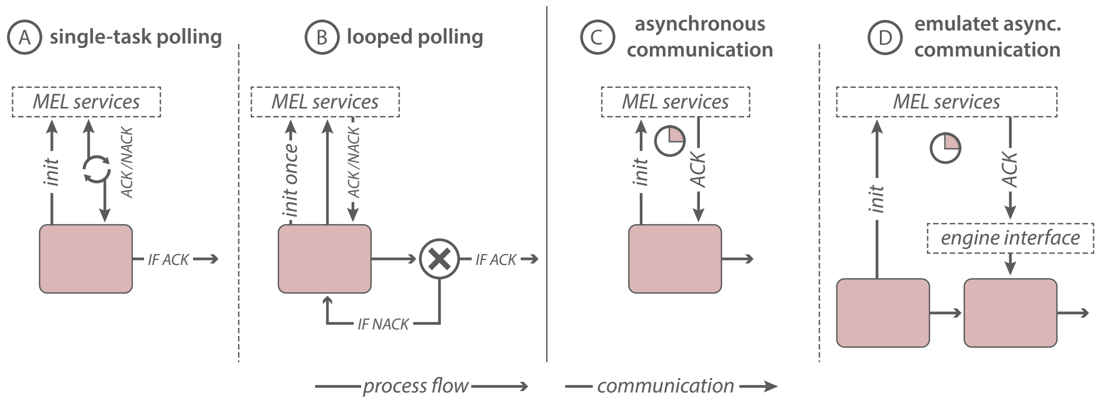


Figure 5.9.: Communication patterns

and termination of mobile tasks, at least one of the communication patterns shown in Figure 5.9 has to be supported by the provided process model, while an asynchronous approach is more desirable to save networking resources.

Ⓐ **Single-task polling:** A synchronous communication pattern between the BPMS and the *MEL* services handled by a single, automated task instance. On activation, the instance sends an initiation request to the *activation service*. After handling the initiation response, the task starts a polling loop, sending recurring requests to the *task finished service*. On obtaining a positive response (i.e., the task has been finished), the loop will be left, the task's data fields and status will be written and the task instance will be marked as valuated.

Ⓑ **Looped polling:** This pattern represents an implementation of Ⓐ using standard features of process control flows. It can be used if a native implementation of the polling behavior is not available. To avoid multiple initiation attempts of the same task instance, an initiation flag has to be set which will be checked on each iteration. The XOR gateway can use data fields provided by the backup operation (e.g., the sync flag) to check the current polling status.

Ⓒ **Asynchronous communication:** If a BPMS supports asynchronous service communication (e.g., by providing callback methods), this can be used by the *task execution interface* to implement the back-channel by sending the finishing status of a mobile task as asynchronous callback. By using this method, the polling service can be deactivated.

Ⓓ **Emulated asynchronous communication:** If the BPMS provides a communication interface to control the task execution, this can be used to implement an asynchronous communication as shown in Figure 5.9 Ⓓ. In this case, the *task execution interface* script has to implement the communication interface and is used as asynchronous callback.

An integration scenario IS_{bpms} for a specific BPMS can be described as a tuple $IS_{bpms} = (URA, AUTH, COM)$. An integration becomes impossible as soon as a single member of this tuple is not compatible with the *MEL* implementation. In the following section, integration scenarios for multiple BPMSs will be identified and used to evaluate whether the BPMS can be integrated.

5.3. Evaluating Business Process Management Systems

As a final step in showing the feasibility of the integration concept in Chapter 4, this section will evaluate the *MEL* prototype by applying it to three integration scenarios. The used scenarios will be derived from existing BPMS implementations by analyzing them with regard to the aspects discussed in the last section. The reviewed BPMSs will be the open source projects *activiti* and *JBoss jBPM 6*, and the commercial BPMS *AristaFlow*.

5.3.1. JBoss jBPM 6

JBoss jBPM [41] is an open source project, aiming to provide a full BPM suit for human and automated task execution. It uses BPMN 2.0 as modeling and execution language and provides tools for managing and modeling business processes. The implementation of the execution model is based on java and meant to be extended by using the provided API. All standard BPMN 2.0 functionalities are already implemented and can be used out of the box.

Regarding the integration context, the module *jBPM workbench* provides a REST API [46], which can be used to control the process execution. For REST clients, a java API is provided, which wraps all possible endpoints and the session invocation. Since *jython* supports the integration of java APIs into python scripts, this can be used for the interface implementation. No interfaces are available to access the organization model. *jBPM* supports a variety of relational databases. If the deployment does not use the in-memory database *H2*, bypassing the system for synchronization becomes a vivid option.

For task automation, *jBPM* provides the `WorkItemHandler` interface and a set of implementations, including the `ServiceTaskHandler`. This class supports synchronous communication between a task instance and a REST service. There are no implemented features for an asynchronous HTTP communication. Hence, both, the polling communication patterns and the emulated asynchronous communication are possible.

A summary of all findings regarding this integration scenario is shown in Table 5.7. Based on this, a compatible integration scenario IS_{jbpm} can be found and the integration of mobile task execution using the *MEL* prototype thus is possible.

URA	No interface. By-pass to relational database		✓
$AUTH$	Session handling using the java REST API		✓
COM	*single-task polling*	`ServiceTaskHandler`	✓
	looped polling	`ServiceTaskHandler`	✓
	asynchronous communication	not implemented	×
	emulated async. communication	`ServiceTaskHandler` and REST API	✓

Table 5.7.: Integration scenario for JBoss jBPM (IS_{jbpm})

5.3.2. Activiti

The open source project *activiti* [40] provides a java-based process execution API for BPMN 2.0 processes. The main goal of the project is not to provide a fully blown BPMS, but an execution environment which may be integrated with other java applications. However, multiple tools and extensions are available to deploy a fully functional BPMS.

Concerning the integration requirements, a REST API implemented with java servlets is available which can be deployed on any java application server [47]. The API is a fully functional wrapper of the java API providing *HTTP basic authentication*, full management access to the organization model and full control over process deployment and execution. Invoking web services from a task instance can be realized with a *web service task* [47] [45]. This is an experimental feature, providing a synchronous HTTP communication with external services. As an alternative, *activiti* supports BPMN extensions, which can be used to implement a proprietary HTTP handler which also supports asynchronous communication. However, since the REST interface provides full control over the execution of task instances, the emulated asynchronous communication pattern can be used. This makes a self-implementation of asynchronous communication obsolete.

As shown in Table 5.8, a compatible integration scenario $IS_{activiti}$ can be found using the java API and the REST extension.

URA	REST API provides full user management		✓
$AUTH$	HTTP Basic Authentication		✓
COM	*single-task polling*	WebServiceTask	✓
	looped polling	WebServiceTask	✓
	asynchronous communication	has to be manually implemented	–
	emulated async. communication	WebServiceTask and REST API for task execution	✓

Table 5.8.: Integration scenario for activiti ($IS_{activiti}$)

5.3.3. AristaFlow

Aristaflow [42][48] is a commercial BPMS developed at the University of Ulm. It focuses on the “component-oriented development of adaptive process-oriented enterprise software” [49] and provides a BPMN 2.0 modeling and execution environment [50], an organization model supporting RBAC [45] and an open java API [51]. For the communication with external services from within an activity, *AristaFlow* provides a set of connector plugins.

A SOAP web service provides interfaces to the organization model, the user session management and the task execution management. To authenticate a user, the user’s name, the role and the password is needed. After a successful authentication, a user session, identified by a token which has to be provided in all further communication, is created.

Based on the SOAP web services and provided standard connectors for HTTP and web service calls, the integration scenario shown in Table 5.9 can be identified.

URA	Organization model access via SOAP web service		✓
$AUTH$	Token based authentication via SOAP web service		✓
COM	*single-task polling*	HTTP or WS connector	✓
	looped polling	HTTP or WS connector	✓
	asynchronous communication	no standard connector available	×
	emulated async. communication	HTTP or WS connector and SOAP web service	✓

Table 5.9.: Integration scenario for AristaFlow ($IS_{aristaflow}$)

5.3.4. Conclusion

The integration scenarios investigated in this section have shown the loose coupling of the *MEL* prototype. By supporting a variety of communication patterns, even older systems can be used for integration. Modern BPMSs, which support multiple patterns, have the freedom of choice and may use this to optimize mobile processes for different architectures and environments. The FPIL enables the prototype to handle highly diverse data formats, and makes the need for new interface implementations obsolete. Widely supported protocols and formats as SOAP (cf. *AristaFlow*) can be used without adaption.

6 Summary and Outlook

In the course of this work, a concept for integrating mobile tasks into BPMSs, based on the *mobile task life cycle* [5] has been introduced. The main goal was to provide a method that enables a BPMS executing tasks on mobile devices, considering challenges arising from the mobile context. In particular, aspects as *connectivity*, *battery status* and *location* in respect of a task's *urgency* have to be considered during the execution. The concept follows a generic approach, introducing a decoupled service layer for executing mobile tasks to reduce effort and costs for the integration into an existing IT infrastructure. Additionally, a prototype has been implemented to show the feasibility of this approach.

Chapter 2 made a comprehensive analysis and categorization of related projects in order to position this work in the current research context. Chapter 3 recapped the concepts which had been introduced in earlier research works [5]. In particular, these included the *mobile task transformation*, defining operations during design time to denote a task as mobile executable, the *delegation service*, which introduced an automated delegation process based on a prioritized user list, and the *backup service* for escalation handling. Based on these concepts, a life cycle for mobile tasks had been defined. However, this approach provides only a rudimentary algorithm for user list prioritization and expects full time connectivity of mobile devices. Facing these shortcomings, an extendible prioritization model, considering location and unsound user behavior (e.g., instant shutdowns), and the concept of *offline tasks*, which enables mobile users to go offline, while performing a mobile task without the task getting delegated, have been introduced.

Chapters 4 and 5 represent the main part of this work. In Chapter 4, requirements of a generic integration approach for mobile task execution were postulated, providing interfaces for mobile clients as well as for underlying BPMSs, and leaving existing interfaces untouched to avoid problems with other system components. With these requirements and the *mobile task life cycle* in mind, a three layer architecture has been developed, introducing the *mobile task transformation module* as extension of the standard *process definition tools*, the *Mobile Execution Layer (MEL)* as a service layer between mobile clients and the underlying BPMS, and a detailed concept for a mobile client implementation. A *MEL* prototype, providing REST services and a *Freely Programmable Interface Layer (FPIL)*, based on python scripts, has been implemented to prove the feasibility of such an approach. The main features, implementation details, and an interaction concept for mobile clients were discussed in Chapter 5. Section 5.2 described a set of possible integration scenarios regarding common communication patterns and provided interfaces to analyze the impact on different BPMSs. They were used in Section 5.3 in order to evaluate the prototype against three real BPMSs: *JBoss JBPM 6*, *Activiti* and *AristaFlow*.

6.1. Conclusion and Future Work

The main goal of this work was to show the feasibility of the concepts introduced by Pryss, Musiol, and Reichert [5]. This was achieved by introducing a comprehensive integration concept for mobile task execution into an existing business process environment, and by the *MEL* prototype implementation. Moreover, the basic concepts were extended to remedy shortcomings, such as offline task execution and user list prioritization. The prototype evaluation shows the feasibility of the proposed concepts and their small impact on the existing IT infrastructure. Hence, this builds the foundation for further research topics.

So far, the prototype does not implement all functionalities described by the integration concept. The tracking of process instances may open new possibilities for monitoring, mining as well as optimization and thus has to be discussed in more detail. Furthermore, a comprehensive evaluation of the *mobile task transformation* and a proof of concept for the *mobile task transformation module* is necessary. It is also possible to integrate other aspects of the business process environment. For example, in [27], the mobile task execution with respect to entailment constraints was discussed. Another possibility could be the combination of this integration concept and related approaches (e.g., fragmentation). Besides the technical aspects, the usability of client applications, which has not been discussed yet, is crucial for the acceptance of an IT system. Finally, the current prototype has to be evaluated and optimized with regards to security, stability and scalability in order to use it in a productive environment.

A
Additional Figures

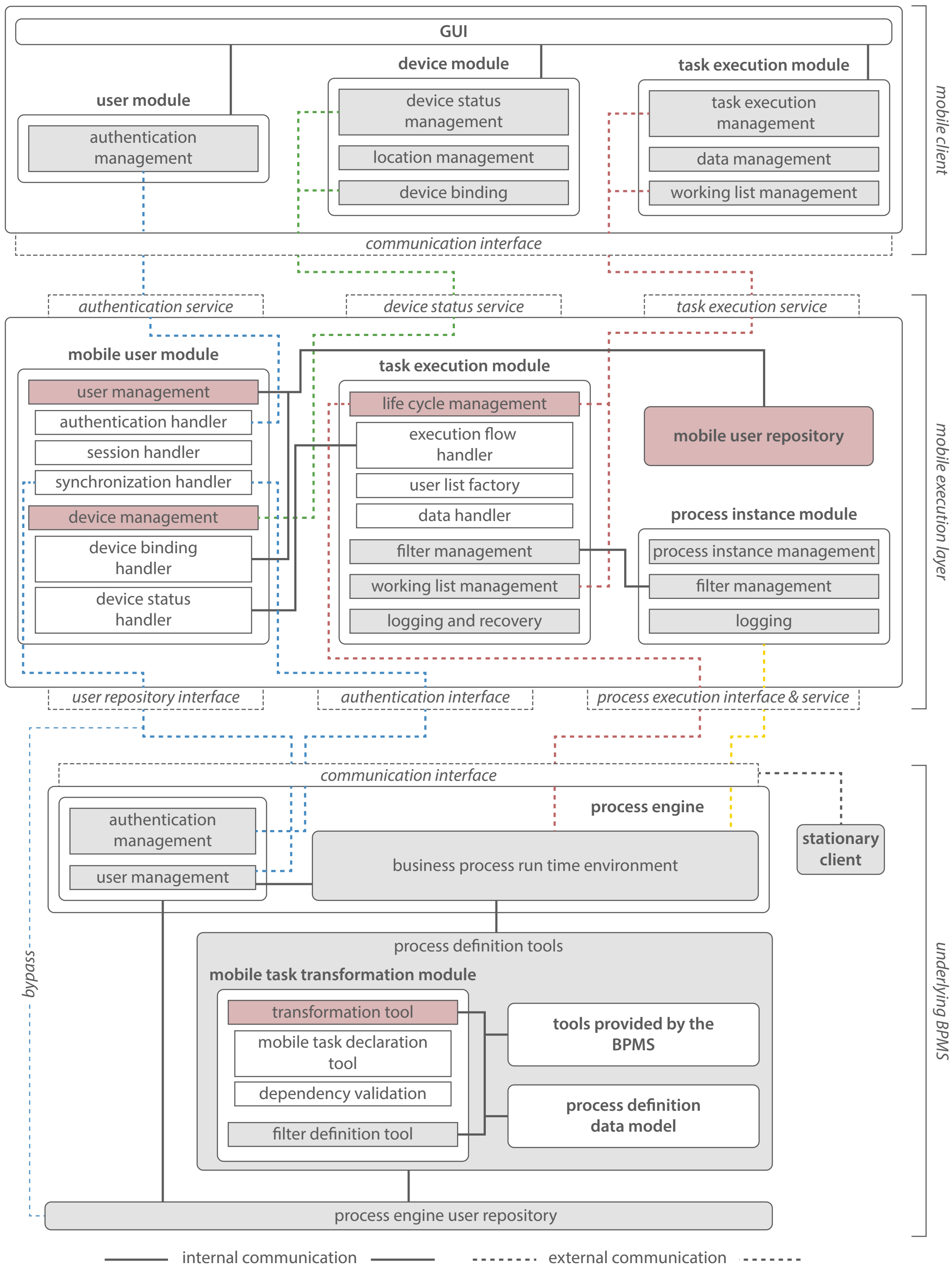


Figure A.1.: Detailed integration concept

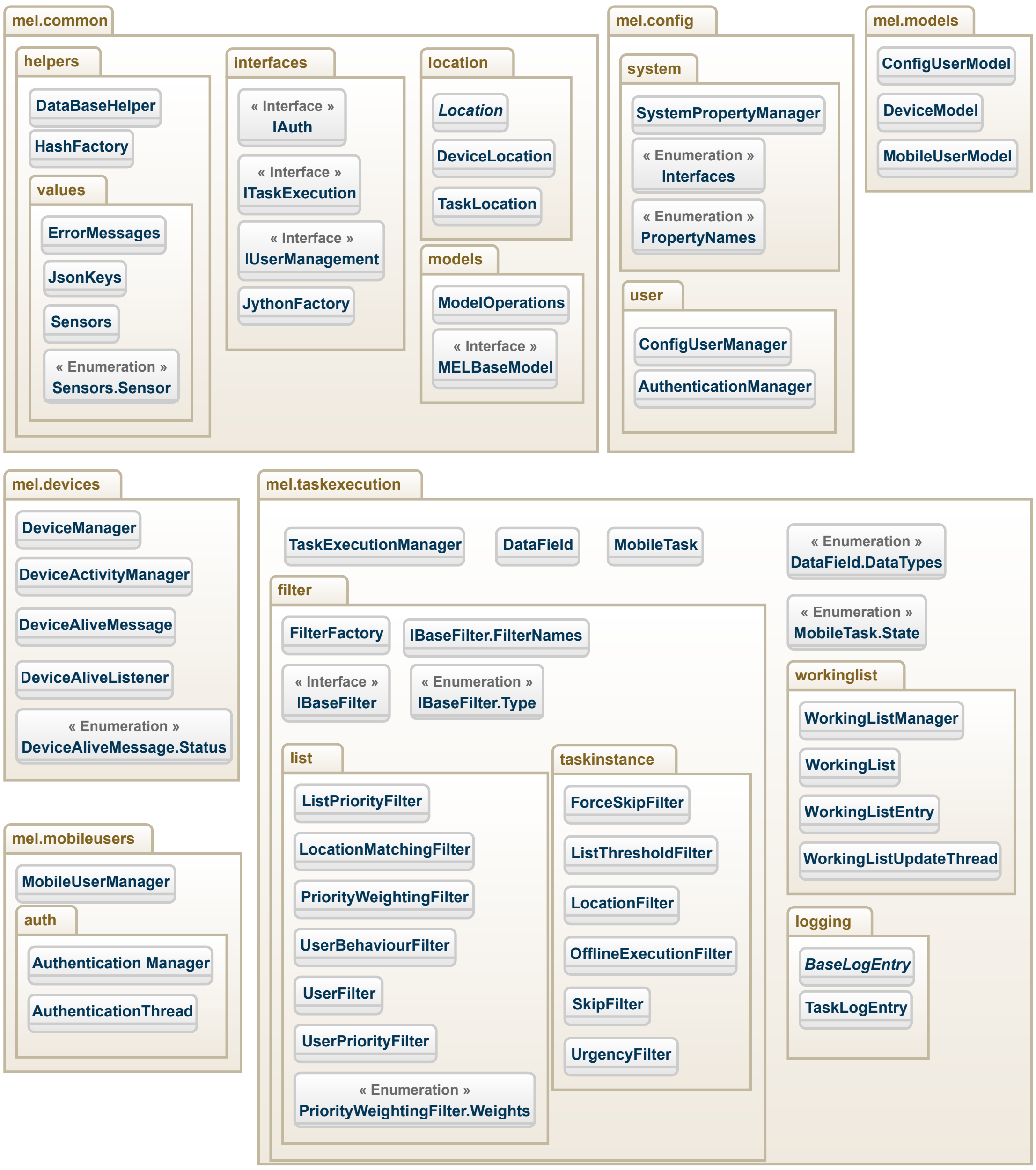


Figure A.2.: Package and class overview

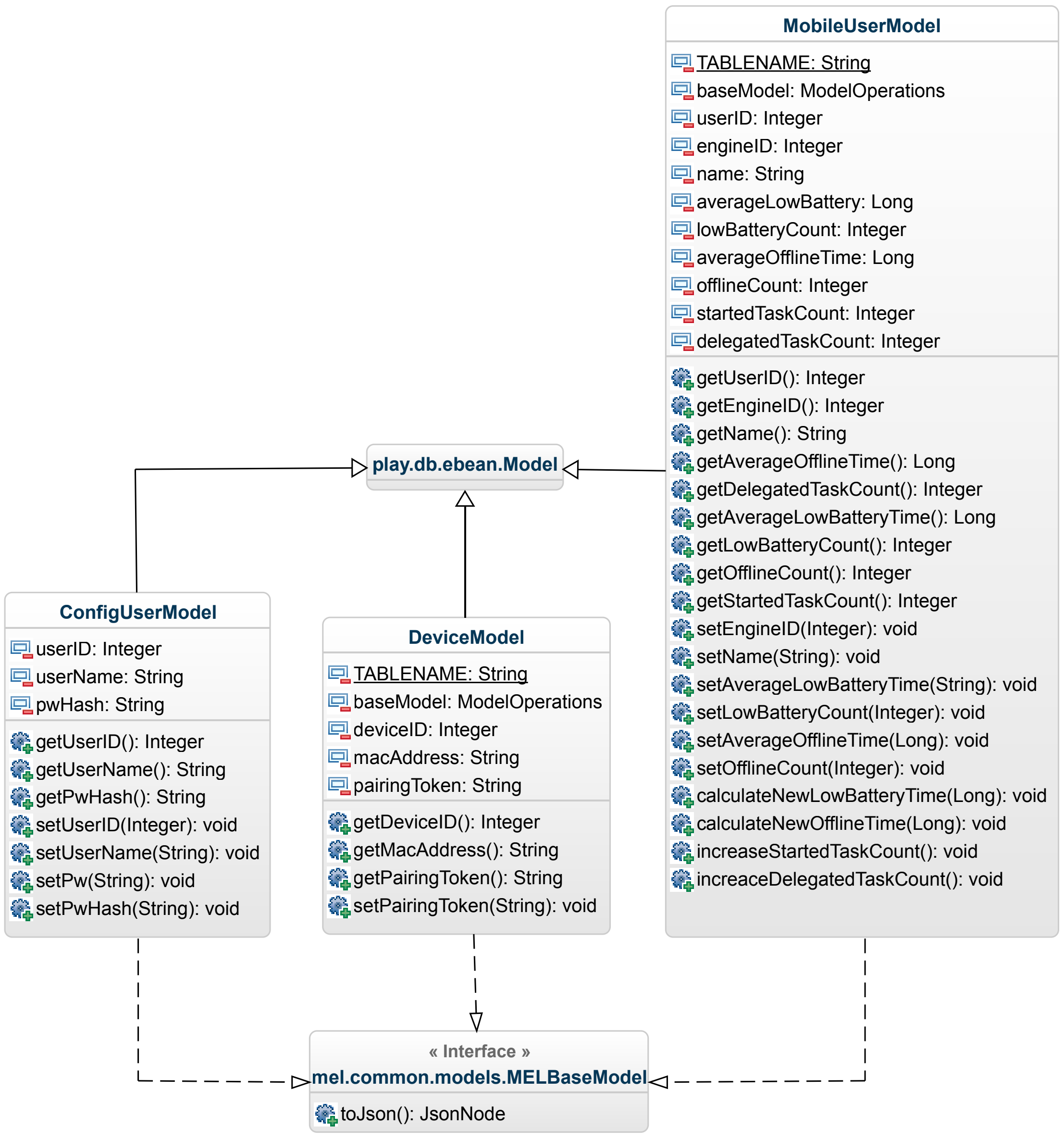

MobileUserModel
TABLENAME: String
baseModel: ModelOperations
userID: Integer
engineID: Integer
name: String
averageLowBattery: Long
lowBatteryCount: Integer
averageOfflineTime: Long
offlineCount: Integer
startedTaskCount: Integer
delegatedTaskCount: Integer
getUserID(): Integer
getEngineID(): Integer
getName(): String
getAverageOfflineTime(): Long
getDelegatedTaskCount(): Integer
getAverageLowBatteryTime(): Long
getLowBatteryCount(): Integer
getOfflineCount(): Integer
getStartedTaskCount(): Integer
setEngineID(Integer): void
setName(String): void
setAverageLowBatteryTime(String): void
setLowBatteryCount(Integer): void
setAverageOfflineTime(Long): void
setOfflineCount(Integer): void
calculateNewLowBatteryTime(Long): void
calculateNewOfflineTime(Long): void
increaseStartedTaskCount(): void
increaceDelegatedTaskCount(): void
play.db.ebean.Model
ConfigUserModel
userID: Integer
userName: String
pwHash: String
getUserID(): Integer
getUserName(): String
getPwHash(): String
setUserID(Integer): void
setUserName(String): void
setPw(String): void
setPwHash(String): void
DeviceModel
TABLENAME: String
baseModel: ModelOperations
deviceID: Integer
macAddress: String
pairingToken: String
getDeviceID(): Integer
getMacAddress(): String
getPairingToken(): String
setPairingToken(String): void
« Interface »
mel.common.models.MELBaseModel
toJson(): JsonNode

Figure A.3.: Package mel.models

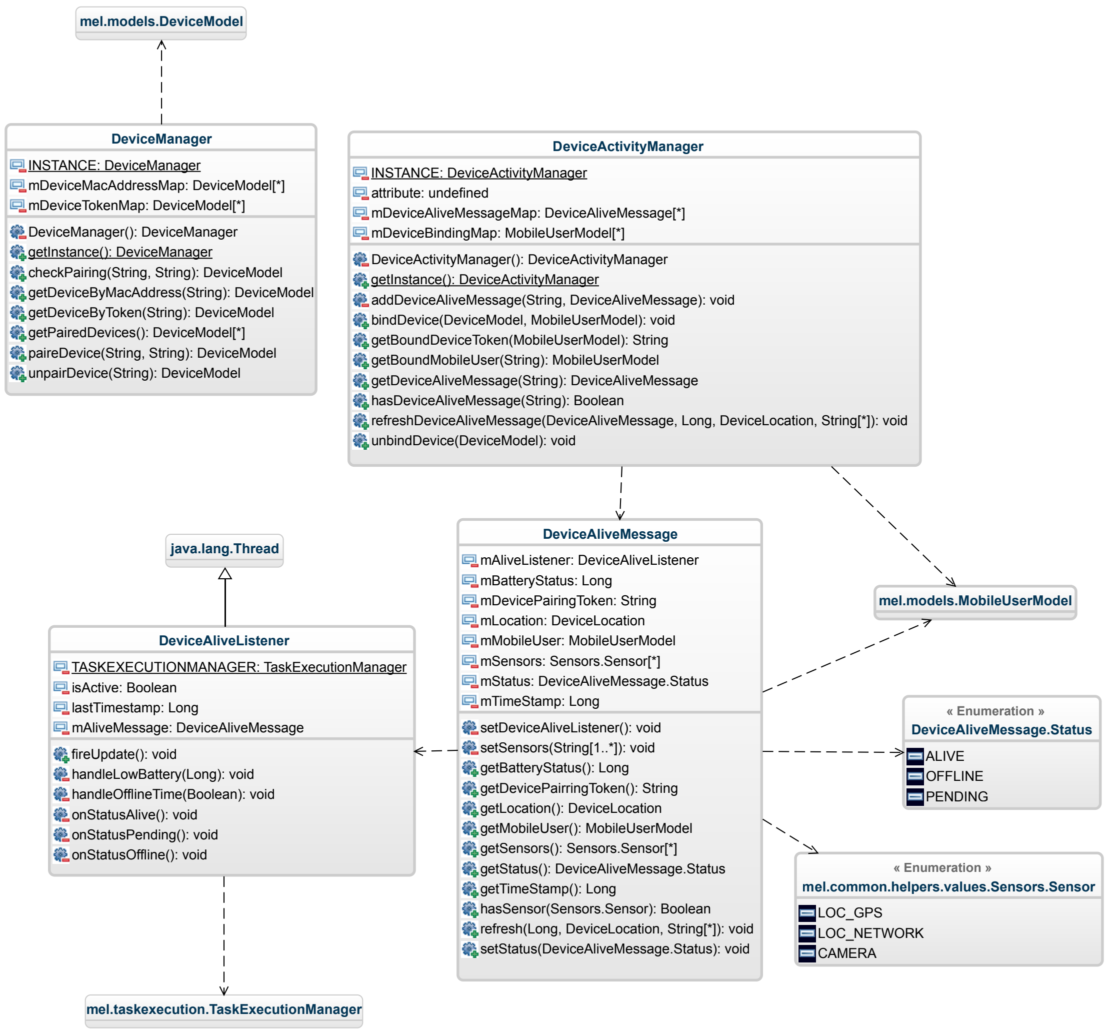

mel.models.DeviceModel
DeviceManager
INSTANCE: DeviceManager
mDeviceMacAddressMap: DeviceModel[*]
mDeviceTokenMap: DeviceModel[*]
DeviceManager(): DeviceManager
getInstance(): DeviceManager
checkPairing(String, String): DeviceModel
getDeviceByMacAddress(String): DeviceModel
getDeviceByToken(String): DeviceModel
getPairedDevices(): DeviceModel[*]
paireDevice(String, String): DeviceModel
unpairDevice(String): DeviceModel
DeviceActivityManager
INSTANCE: DeviceActivityManager
attribute: undefined
mDeviceAliveMessageMap: DeviceAliveMessage[*]
mDeviceBindingMap: MobileUserModel[*]
DeviceActivityManager(): DeviceActivityManager
getInstance(): DeviceActivityManager
addDeviceAliveMessage(String, DeviceAliveMessage): void
bindDevice(DeviceModel, MobileUserModel): void
getBoundDeviceToken(MobileUserModel): String
getBoundMobileUser(String): MobileUserModel
getDeviceAliveMessage(String): DeviceAliveMessage
hasDeviceAliveMessage(String): Boolean
refreshDeviceAliveMessage(DeviceAliveMessage, Long, DeviceLocation, String[*]): void
unbindDevice(DeviceModel): void
java.lang.Thread
DeviceAliveListener
TASKEXECUTIONMANAGER: TaskExecutionManager
isActive: Boolean
lastTimestamp: Long
mAliveMessage: DeviceAliveMessage
fireUpdate(): void
handleLowBattery(Long): void
handleOfflineTime(Boolean): void
onStatusAlive(): void
onStatusPending(): void
onStatusOffline(): void
mel.taskexecution.TaskExecutionManager
DeviceAliveMessage
mAliveListener: DeviceAliveListener
mBatteryStatus: Long
mDevicePairingToken: String
mLocation: DeviceLocation
mMobileUser: MobileUserModel
mSensors: Sensors.Sensor[*]
mStatus: DeviceAliveMessage.Status
mTimeStamp: Long
setDeviceAliveListener(): void
setSensors(String[1..*]): void
getBatteryStatus(): Long
getDevicePairringToken(): String
getLocation(): DeviceLocation
getMobileUser(): MobileUserModel
getSensors(): Sensors.Sensor[*]
getStatus(): DeviceAliveMessage.Status
getTimeStamp(): Long
hasSensor(Sensors.Sensor): Boolean
refresh(Long, DeviceLocation, String[*]): void
setStatus(DeviceAliveMessage.Status): void
mel.models.MobileUserModel
« Enumeration »
DeviceAliveMessage.Status
ALIVE
OFFLINE
PENDING
« Enumeration »
mel.common.helpers.values.Sensors.Sensor
LOC_GPS
LOC_NETWORK
CAMERA

Figure A.4.: Package mel.devices

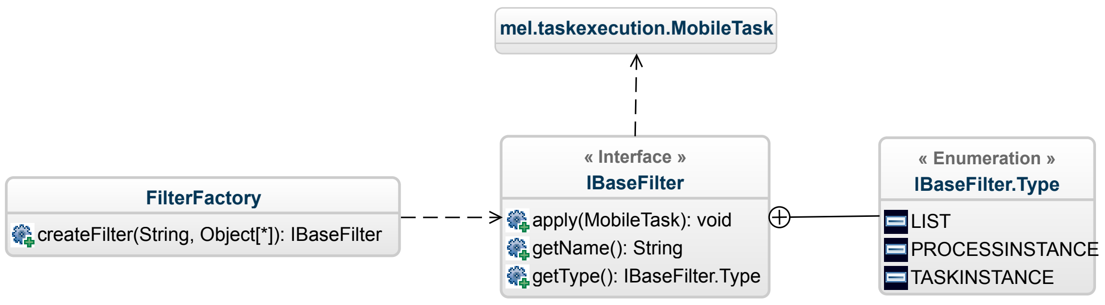


Figure A.5.: Package mel.taskexecution.filters

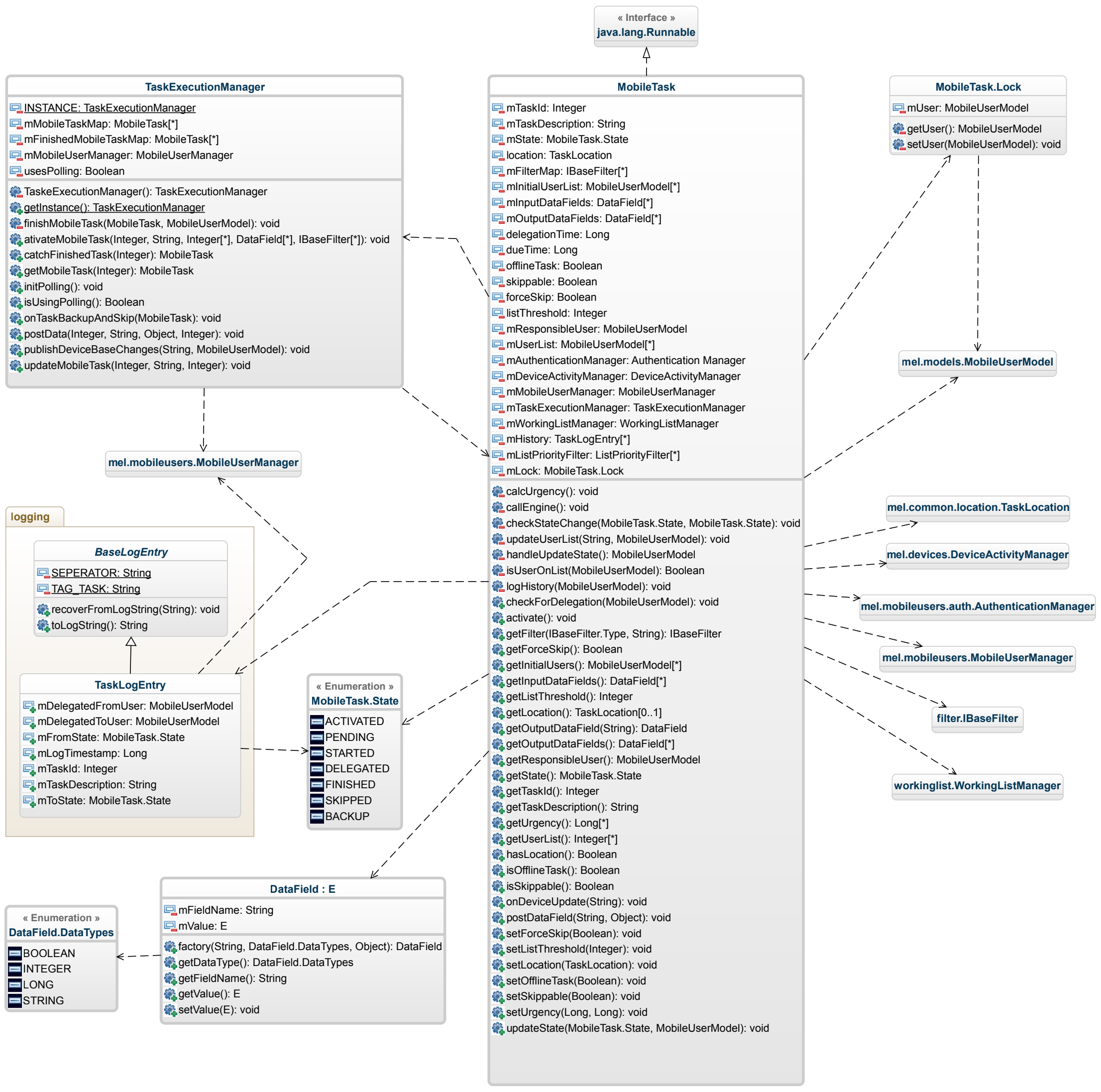

« Interface »
java.lang.Runnable
TaskExecutionManager
INSTANCE: TaskExecutionManager
mMobileTaskMap: MobileTask[*]
mFinishedMobileTaskMap: MobileTask[*]
mMobileUserManager: MobileUserManager
usesPolling: Boolean
TaskeExecutionManager(): TaskExecutionManager
getInstance(): TaskExecutionManager
finishMobileTask(MobileTask, MobileUserModel): void
ativateMobileTask(Integer, String, Integer[*], DataField[*], IBaseFilter[*]): void
catchFinishedTask(Integer): MobileTask
getMobileTask(Integer): MobileTask
initPolling(): void
isUsingPolling(): Boolean
onTaskBackupAndSkip(MobileTask): void
postData(Integer, String, Object, Integer): void
publishDeviceBaseChanges(String, MobileUserModel): void
updateMobileTask(Integer, String, Integer): void
MobileTask
mTaskId: Integer
mTaskDescription: String
mState: MobileTask.State
location: TaskLocation
mFilterMap: IBaseFilter[*]
mInitialUserList: MobileUserModel[*]
mInputDataFields: DataField[*]
mOutputDataFields: DataField[*]
delegationTime: Long
dueTime: Long
offlineTask: Boolean
skippable: Boolean
forceSkip: Boolean
listThreshold: Integer
mResponsibleUser: MobileUserModel
mUserList: MobileUserModel[*]
mAuthenticationManager: Authentication Manager
mDeviceActivityManager: DeviceActivityManager
mMobileUserManager: MobileUserManager
mTaskExecutionManager: TaskExecutionManager
mWorkingListManager: WorkingListManager
mHistory: TaskLogEntry[*]
mListPriorityFilter: ListPriorityFilter[*]
mLock: MobileTask.Lock
calcUrgency(): void
callEngine(): void
checkStateChange(MobileTask.State, MobileTask.State): void
updateUserList(String, MobileUserModel): void
handleUpdateState(): MobileUserModel
isUserOnList(MobileUserModel): Boolean
logHistory(MobileUserModel): void
checkForDelegation(MobileUserModel): void
activate(): void
getFilter(IBaseFilter.Type, String): IBaseFilter
getForceSkip(): Boolean
getInitialUsers(): MobileUserModel[*]
getInputDataFields(): DataField[*]
getListThreshold(): Integer
getLocation(): TaskLocation[0..1]
getOutputDataField(String): DataField
getOutputDataFields(): DataField[*]
getResponsibleUser(): MobileUserModel
getState(): MobileTask.State
getTaskId(): Integer
getTaskDescription(): String
getUrgency(): Long[*]
getUserList(): Integer[*]
hasLocation(): Boolean
isOfflineTask(): Boolean
isSkippable(): Boolean
onDeviceUpdate(String): void
postDataField(String, Object): void
setForceSkip(Boolean): void
setListThreshold(Integer): void
setLocation(TaskLocation): void
setOfflineTask(Boolean): void
setSkippable(Boolean): void
setUrgency(Long, Long): void
updateState(MobileTask.State, MobileUserModel): void
MobileTask.Lock
mUser: MobileUserModel
getUser(): MobileUserModel
setUser(MobileUserModel): void
mel.models.MobileUserModel
mel.common.location.TaskLocation
mel.devices.DeviceActivityManager
mel.mobileusers.auth.AuthenticationManager
mel.mobileusers.MobileUserManager
filter.IBaseFilter
workinglist.WorkingListManager
mel.mobileusers.MobileUserManager
logging
BaseLogEntry
SEPERATOR: String
TAG_TASK: String
recoverFromLogString(String): void
toLogString(): String
TaskLogEntry
mDelegatedFromUser: MobileUserModel
mDelegatedToUser: MobileUserModel
mFromState: MobileTask.State
mLogTimestamp: Long
mTaskId: Integer
mTaskDescription: String
mToState: MobileTask.State
« Enumeration »
MobileTask.State
ACTIVATED
PENDING
STARTED
DELEGATED
FINISHED
SKIPPED
BACKUP
DataField : E
mFieldName: String
mValue: E
factory(String, DataField.DataTypes, Object): DataField
getDataType(): DataField.DataTypes
getFieldName(): String
getValue(): E
setValue(E): void
« Enumeration »
DataField.DataTypes
BOOLEAN
INTEGER
LONG
STRING

Figure A.6.: Package mel.taskexecution

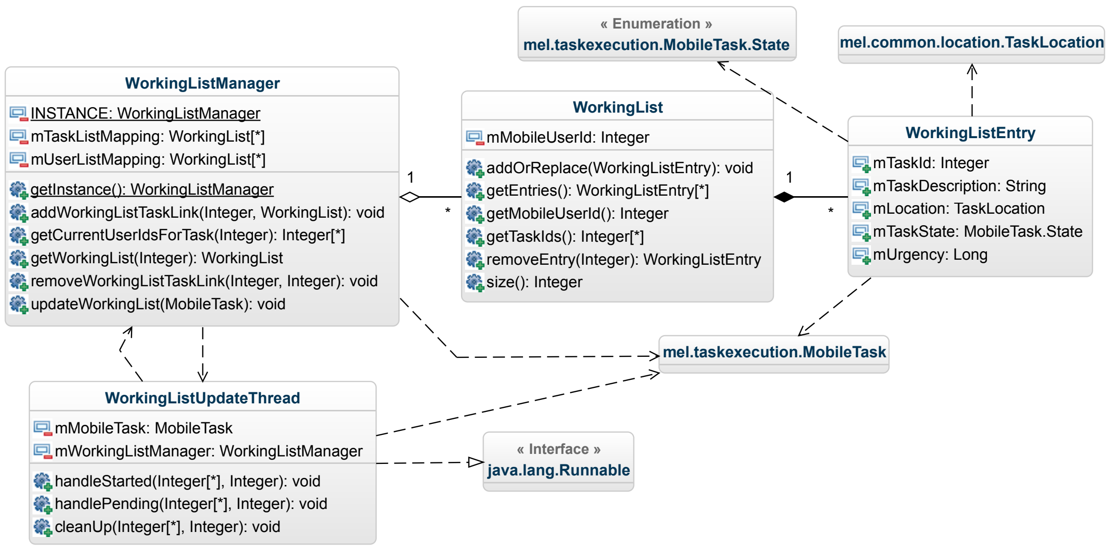

« Enumeration »
mel.taskexecution.MobileTask.State
mel.common.location.TaskLocation
WorkingListManager
INSTANCE: WorkingListManager
mTaskListMapping: WorkingList[*]
mUserListMapping: WorkingList[*]
getInstance(): WorkingListManager
addWorkingListTaskLink(Integer, WorkingList): void
getCurrentUserIdsForTask(Integer): Integer[*]
getWorkingList(Integer): WorkingList
removeWorkingListTaskLink(Integer, Integer): void
updateWorkingList(MobileTask): void
1
*
WorkingList
mMobileUserId: Integer
addOrReplace(WorkingListEntry): void
getEntries(): WorkingListEntry[*]
getMobileUserId(): Integer
getTaskIds(): Integer[*]
removeEntry(Integer): WorkingListEntry
size(): Integer
1
*
WorkingListEntry
mTaskId: Integer
mTaskDescription: String
mLocation: TaskLocation
mTaskState: MobileTask.State
mUrgency: Long
mel.taskexecution.MobileTask
WorkingListUpdateThread
mMobileTask: MobileTask
mWorkingListManager: WorkingListManager
handleStarted(Integer[*], Integer): void
handlePending(Integer[*], Integer): void
cleanUp(Integer[*], Integer): void
« Interface »
java.lang.Runnable

Figure A.7.: Package mel.taskexecution.workinglists

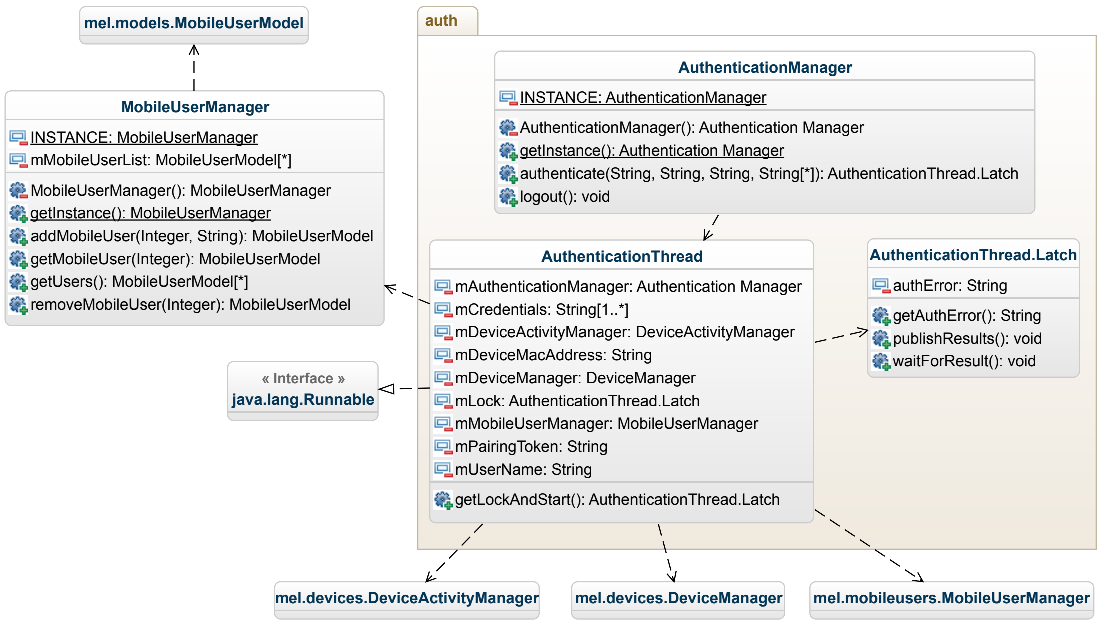

mel.models.MobileUserModel
MobileUserManager
INSTANCE: MobileUserManager
mMobileUserList: MobileUserModel[*]
MobileUserManager(): MobileUserManager
getInstance(): MobileUserManager
addMobileUser(Integer, String): MobileUserModel
getMobileUser(Integer): MobileUserModel
getUsers(): MobileUserModel[*]
removeMobileUser(Integer): MobileUserModel
auth
AuthenticationManager
INSTANCE: AuthenticationManager
AuthenticationManager(): Authentication Manager
getInstance(): Authentication Manager
authenticate(String, String, String, String[*]): AuthenticationThread.Latch
logout(): void
AuthenticationThread
mAuthenticationManager: Authentication Manager
mCredentials: String[1..*]
mDeviceActivityManager: DeviceActivityManager
mDeviceMacAddress: String
mDeviceManager: DeviceManager
mLock: AuthenticationThread.Latch
mMobileUserManager: MobileUserManager
mPairingToken: String
mUserName: String
getLockAndStart(): AuthenticationThread.Latch
AuthenticationThread.Latch
authError: String
getAuthError(): String
publishResults(): void
waitForResult(): void
« Interface »
java.lang.Runnable
mel.devices.DeviceActivityManager
mel.devices.DeviceManager
mel.mobileusers.MobileUserManager

Figure A.8.: Package mel.mobileusers

B

REST Service Index

Mobile User Authentication

POST /auth/login

Parameters: <none>

Payload:

```
{ "userName" : String ,
  "macAddress" : String ,
  "pairingToken" : String ,
  "credentials" : String[] }
```

HTTP 200 <empty>
HTTP 400 <Missing Parameter>
HTTP 401 <Error Message>
HTTP 500 <Error Message>

GET /auth/logout

Parameters: <none>	**HTTP 200** <empty>
Payload: <empty>	**HTTP 401** <Error Message>
	HTTP 500 <Error Message>

POST /auth/dam

Parameters: <none>

Payload:

```
{ "pairingToken" : String ,
  "macAddress" : String ,
  "batteryStatus" : Long,
  "location" : {
    "x" : Long,
    "y" : Long
    },
  "sensors" : String[] }
```

HTTP 200 <empty>
HTTP 400 <Missing Parameter>
HTTP 401 <Error Message>
HTTP 500 <Error Message>

Mobile Task Execution

GET /workinglist

Parameters: <none>
Payload: <empty>

HTTP 200

```
{ "workingList" : [
  { "taskId" : Integer ,
    "taskDescription" : String ,
    "taskStatus" : String }]}
```

HTTP 401 <Error Message>
HTTP 500 <Error Message>

GET /task/*<taskId>*

Parameters: <none>

Payload: <empty>

HTTP 200

```
{ "taskId" : Integer ,
  "taskDescription" : String ,
  "inputData" : [
    { "dataFieldName" : String ,
      "dataType" : String ,
      "dataValue" : String }] ,
  "outputData" : [
    { "dataFieldName" : String ,
      "dataType" : String ,
      "dataValue" : <dataType> }]}
```

HTTP 400 <No Task>

HTTP 401 <Error Message>

HTTP 500 <Error Message>

POST /task/*<taskId>*

Parameters: <none>

Payload: <empty>

HTTP 200

```
{ "taskId" : Integer ,
  "taskDescription" : String ,
  "inputData" : [
    { "dataFieldName" : String ,
      "dataType" : String ,
      "dataValue" : String }] ,
  "outputData" : [
    { "dataFieldName" : String ,
      "dataType" : String ,
      "dataValue" : <dataType> }]}
```

HTTP 400 <No Task>

HTTP 401 <Error Message>

HTTP 500 <Error Message>

PUT /task/*<taskId>*

Parameters: <none>

Payload:

```
{ "dataFieldName" : String ,
    "dataType" : String ,
    "dataValue" : <dataType> }
```

HTTP 200

```
{ "taskId" : Integer ,
  "taskDescription" : String ,
  "inputData" : [
    { "dataFieldName" : String ,
      "dataType" : String ,
      "dataValue" : String }] ,
  "outputData" : [
    { "dataFieldName" : String ,
      "dataType" : String ,
      "dataValue" : <dataType> }]}
```

HTTP 400 <Error Message>

HTTP 401 <Error Message>

HTTP 500 <Error Message>

DELETE /task/*<taskId>*

Parameters: <none>

Payload: <empty>

HTTP 200

```
{ "taskId" : Integer ,
  "taskDescription" : String ,
  "inputData" : [
    { "dataFieldName" : String ,
      "dataType" : String ,
      "dataValue" : String }] ,
  "outputData" : [
    { "dataFieldName" : String ,
      "dataType" : String ,
      "dataValue" : <dataType> }]}
```

HTTP 400 <No Task>

HTTP 401 <Error Message>

HTTP 500 <Error Message>

Engine Services

POST /engine/task/*<taskId>*

Parameters: <none>

Payload:

```
{ "taskId" : Integer ,
  "taskDescription" : String ,
  "userList" : Integer[] ,
  "inputData" : [
    { "dataFieldName" : String ,
      "dataType" : String ,
      "dataValue" : String }] ,
  "outputData" : [
    { "dataFieldName" : String ,
      "dataType" : String ,
      "dataValue" : String }] ,
  "executionFilters" : [
    { "filterName" : String ,
      "filterValues" : [] }]}
```

HTTP 200 <empty>

HTTP 400 <Missing Parameter>

HTTP 401 <Error Message>

HTTP 500 <Error Message>

GET /engine/task/*<taskId>*

Parameters: <none>

Payload: <empty>

HTTP 200

```
{ "taskId" : Integer ,
  "taskDescription" : String ,
  "outputData" : [
    { "dataFieldName" : String ,
      "dataType" : String ,
      "dataValue" : <dataType> }]}
```

HTTP 400 <No Task>

HTTP 401 <Error Message>

HTTP 500 <Error Message>

Configuration User Authentication

POST /config/authrequest

Parameters:
token : String

Payload:
```
{ "userName" : String ,
  "reqHash" : String ,
  "anonce" : String
}
```

HTTP 200
```
{ "userName" : String ,
  "kNonce": String }
```

HTTP 400 <Missing Parameter>

HTTP 401 <Error Message>

HTTP 500 <Error Message>

POST /config/authfinal

Parameters:
token : String

Payload:
```
{ "userName" : String ,
  "token" : String
}
```

HTTP 200
```
{ "userName" : String ,
  "token": String }
```

HTTP 400 <Missing Parameter>

HTTP 401 <Error Message>

HTTP 500 <Error Message>

GET /config/logout

Parameters:
token : String

Payload: <empty>

HTTP 200 <empty>

HTTP 401 <Error Message>

HTTP 500 <Error Message>

Configuration User Management

GET /config/configusers

Parameters:

token : String

Payload: <empty>

HTTP 200

```
{ "token" : String,
  "cfgUser":[{
    "userID":Integer,
    "userName":String,
    "pwHash":String
   }] }
```

HTTP 401 <Error Message>

HTTP 500 <Error Message>

GET /config/configusers/*<id>*

Parameters:

token : String

Payload: <empty>

HTTP 200

```
{ "token" : String,
  "cfgUser":{
    "userID":<id> Integer,
    "userName":String,
    "pwHash":String } }
```

HTTP 401 <Error Message>

HTTP 500 <Error Message>

POST /config/configusers

Parameters:

token : String

Payload:

```
{ "userName" : String,
  "password" : String }
```

HTTP 200

```
{ "token" : String,
  "cfgUser":{
    "userID":<id> Integer,
    "userName":String,
    "pwHash": String } }
```

HTTP 400 <Missing Parameter>
HTTP 401 <Error Message>
HTTP 500 <Error Message>

PUT /config/configusers/*<id>*

Parameters:	HTTP 200
token : String	{ "token" : String ,
Payload:	"cfgUser":{
{ "userName" : String ,	"userID":<id> Integer ,
"password" : String }	"userName": String ,
	"pwHash": String } }
	HTTP 400 <Error Message>
	HTTP 401 <Error Message>
	HTTP 500 <Error Message>

DELETE /config/configusers/*<id>*

Parameters:	HTTP 200
token : String	{ "token" : String ,
Payload: <empty>	"cfgUser":{
	"userID":<id> Integer ,
	"userName": String ,
	"pwHash": String } }
	HTTP 400 <Error Message>
	HTTP 401 <Error Message>
	HTTP 500 <Error Message>

Mobile User Management

GET /config/mobileusers/import

Parameters:
token : String

Payload: <empty>

HTTP 200

```
{ "token" : String ,
  "engineUsers" : {
    "users" : [ {
      "engineID" : Integer ,
      "username" : String ,
      "name" : String }]}}
```

HTTP 401 <Error Message>

HTTP 500 <Error Message>

GET /config/mobileusers

Parameters:
token : String

Payload: <empty>

HTTP 200

```
{ "token" : String ,
  "mobileUsers" : [{
    "userID" : Integer ,
    "engineID" : Integer ,
    "name" : String ,
    "offlineCount" : Integer ,
    "lowBatteryCount" : Integer ,
    "averageOfflineTime" : Integer ,
    "averageLowBatteryTime" : Integer ,
    "startedTaskCount" : Integer ,
    "delegatedTaskCount" : Integer }]}
```

HTTP 401 <Error Message>

HTTP 500 <Error Message>

GET /config/mobileusers/*<id>*

Parameters:

token : String

Payload: <empty>

HTTP 200

```
{ "token" : String ,
  "mobUser" : {
    "userID" : Integer ,
    "engineID" : Integer ,
    "name" : String ,
    "offlineCount" : Integer ,
    "lowBatteryCount" : Integer ,
    "averageOfflineTime" : Integer ,
    "averageLowBatteryTime" : Integer ,
    "startedTaskCount" : Integer ,
    "delegatedTaskCount" : Integer }}
```

HTTP 401 <Error Message>

HTTP 500 <Error Message>

POST /config/mobileusers

Parameters:

token : String

Payload:

```
{ "engineID" : Integer ,
  "name" : String }
```

HTTP 200

```
{ "token" : String ,
  "mobUser" : {
    "userID" : Integer ,
    "engineID" : Integer ,
    "name" : String ,
    "offlineCount" : Integer ,
    "lowBatteryCount" : Integer ,
    "averageOfflineTime" : Integer ,
    "averageLowBatteryTime" : Integer ,
    "startedTaskCount" : Integer ,
    "delegatedTaskCount" : Integer }}
```

HTTP 400 <Missing Parameter>

HTTP 401 <Error Message>

HTTP 500 <Error Message>

DELETE /config/mobileusers/*<id>*

Parameters:

token : String

Payload: <empty>

HTTP 200

```
{ "token" : String ,
  "mobUser" : {
    "userID" : Integer ,
    "engineID" : Integer ,
    "name" : String ,
    "offlineCount" : Integer ,
    "lowBatteryCount" : Integer ,
    "averageOfflineTime" : Integer ,
    "averageLowBatteryTime" : Integer ,
    "startedTaskCount" : Integer ,
    "delegatedTaskCount" : Integer }}
```

HTTP 401 <Error Message>

HTTP 500 <Error Message>

Mobile Device Management

GET /config/devices

Parameters:
token : String

Payload: <empty>

HTTP 200
```
{ "token" : String ,
  "devices" : [{
    "deviceID" : Integer ,
    "macAddress" : String ,
    "pairingToken" : String }]}
```

HTTP 401 <Error Message>

HTTP 500 <Error Message>

GET /config/devices/*<pairingToken>*

Parameters:
token : String

Payload: <empty>

HTTP 200
```
{ "token" : String ,
  "device" : {
    "deviceID" : Integer ,
    "macAddress" : String ,
    "pairingToken" : String }}
```

HTTP 401 <Error Message>

HTTP 500 <Error Message>

POST /config/devices

Parameters:
token : String

Payload:
```
{ "macAddress" : String }
```

HTTP 200
```
{ "token" : String ,
  "device" : {
    "deviceID" : Integer ,
    "macAddress" : String ,
    "pairingToken" : String }}
```

HTTP 400 <Missing Parameter>

HTTP 401 <Error Message>

HTTP 500 <Error Message>

DELETE /config/devices/*<pairingToken>*

Parameters:	**HTTP 200**
token : String	{ "token" : String ,
Payload: <empty>	"device" : {
	"deviceID" : Integer ,
	"macAddress" : String ,
	"pairingToken" : String }}
	HTTP 401 <Error Message>
	HTTP 500 <Error Message>

C

Prototype Installation and Configuration

Installation

The prototype requires a MySQL database installation. Before installing the application, make sure, you have a database in place.

To install the application on a server, the following steps have to be taken:

1. Download and install the play v 2.1.2.
   (`http://downloads.typesafe.com/play/2.1.2/play-2.1.2.zip`)
2. Copy the *source*-folder to the server (the location can be freely chosen).
3. Edit the */conf/application.conf* file as follows:
   a) Replace `<dbname>` in *db.default.url* and *db.default.jndiName* with the name of your database.

   b) Replace `<host>` in *db.default.url* with your database host name.
   c) Replace `<dbuser>` in *db.default.user* with the database user.
   d) Replase `<password>` in *db.default.password* with the respective password

4. Open a terminal in the application folder and run the `play run` command.
5. After the application started, the services are available at port 9000. Open up a browser and query an arbitrary service (e.g., `/config/authrequest`). Confirm the dialog to apply the initial SQL script.
6. The application is now installed and all services can be used. An initial configuration user *admin* with the password *pass* was created.

Configuration User Authentication

The authentication process for the configuration services needs a two step authentication. For all further service calls, a security token, which is calculated during the authentication, has to be provided as url parameter (i.e., `?token=<token>`). Each token has a *time to live (TTL)*, which can be configured in the configuration file. When when the TTL is over, the server may create a new token and will send it with the response. From now on the new token has to be used. The following steps have to be taken for a successful authentication:

1. The client generates a random `anonce`.
2. The client calculates the `reqHash` with $reqHash = SHA256(password, anonce)$.
3. The client sends the user name, `reqHash` and `anonce` to the server (`endpoint: /config/authrequest`).
4. The Server validates the reqHash and sends a random `knonce` as response
5. Both calculate the session token with $token = SHA256(reqHash, knonce)$.
6. To finish the authentication, the client sends teh token to the `/config/authfinal` endpoint.

Configuration

The service behaviour can be configured in the configuration file */public/settings.properties*. An overview of all possible properties can be found in Table C.1.

Property	Type	Description
config_Nonce_LENGTH	int	Lengh of anonce and knonce. Default: 10
config_Token_TTL	int	TTL for security tokens (in ms). Default: 300000
config_interface_usepolling=0	int	Enables task polling (0=disabled, 1=enabled)
dam_timeout_alive	int	ALIVE timeout for DAMs (in ms). Default: 5000
dam_timeout_pending	int	PENDING timeout for DAMs (in ms). Default: 5000
dam_timeout_offline	int	OFFLINE timeout for DAMs (in ms). Devices will be unbounded. Default: 5000
dam_lowbattery_threshold	int	% when a battery status is considered as *low*. Default: 10
interface_root	String	Interface root folder
interface_folder_auth	String	Authentication scripts folder
interface_folder_taskexecution	String	Task execution scripts folder
interface_folder_taskactivation	String	Task activation scripts folder
interface_folder_usermanagement	String	User import scripts folder
interface_AUTH	String	Authentication script (.py)
interface_TASKEXECUTION	String	Task Execution script (.py)
interface_TASKACTIVATION	String	Task Activation script (.py)
interface_USERMANAGEMENT	String	User import script (.py)

Table C.1.: Prototype configuration properties

List of Figures

List of Tables

List of Acronyms

API	Application Programming Interface
BPMS	Business Process Management System
DAM	Device Alive Message
FPIL	Freely Programmable Interface Layer
GUI	Graphical User Interface
HTTP	Hyper Text Transfer Protocol
MANET	Mobile Ad-hoc Network
MDS	Mobile Delegation Service
MEL	Mobile Execution Layer
MVC	Model View Controller
RBAC	Role Based Access Control
REST	Representational State Transfer
SOAP	Simple Object Access Protocol

Bibliography

[1] *State of Mobile 2013 (Infographic) - Super Monitoring Blog*. Sept. 2013. *http://www.supermonitoring.com/blog/2013/09/23/state-of-mobile-2013-infographic/*.

[2] Georg Disterer and Carsten Kleiner. “BYOD — Bring Your Own Device”. In: *HMD Praxis der Wirtschaftsinformatik* 50.2 (2013), pp. 92–100.

[3] Rüdiger Pryss et al. “Supporting Medical Ward Rounds through Mobile Task and Process Management”. In: *Information Systems and e-Business Management* (2014), pp. 1–40.

[4] Thomas Allweyer. *BPMN 2.0: Introduction to the Standard for Business Process Modeling*. BoD–Books on Demand, 2010.

[5] Rüdiger Pryss, Steffen Musiol, and Manfred Reichert. “Extending Business Processes with Mobile Task Support: A Self-Healing Solution Architecture”. In: *Handbook of Research on Architectural Trends in Service-Driven Computing*. IGI Global, 2014.

[6] George H. Forman and John Zahorjan. “The Challenges of Mobile Computing”. In: *Computer* 27.4 (Apr. 1994), 38–47.

[7] M. Satyanarayanan. “Fundamental Challenges in Mobile Computing”. In: *In ACM Symposium on Principles of Distributed Computing*. 1996, 1–7.

[8] Rüdiger Pryss et al. “Mobile Task Management for Medical Ward Rounds – The MEDo Approach”. In: *Business Process Management Workshops*. Ed. by Marcello La Rosa and Pnina Soffer. Lecture Notes in Business Information Processing 132. Springer Berlin Heidelberg, 2013, pp. 43–54.

[9] E. Philips, R. Van Der Straeten, and V. Jonckers. “NOW: Orchestrating Services in a Nomadic Network using a Dedicated Workflow Language”. In: *Science of Computer Programming* 78.2 (Feb. 1, 2013), pp. 168–194.

[10] Eline Philips, Ragnhild Van Der Straeten, and Viviane Jonckers. “NOW: A Workflow Language for Orchestration in Nomadic Networks”. In: *Coordination Models and Languages*. Ed. by Dave Clarke and Gul Agha. Lecture Notes in Computer Science 6116. Springer Berlin Heidelberg, 2010, pp. 31–45.

[11] Gregory Hackmann, Christopher Gill, and Gruia-catalin Roman. “Extending BPEL for Interoperable Pervasive Computing”. In: *Pervasive Services, IEEE International Conference on*. IEEE. 2007, pp. 204–213.

[12] *Web Services Business Process Execution Language*. Apr. 11, 2007. *http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html*.

[13] Anna Kocurova et al. “Context-Aware Content-Centric Collaborative Workflow Management for Mobile Devices”. In: COLLA 2012, The Second International Conference on Advanced Collaborative Networks, Systems and Applications. June 24, 2012, pp. 54–57.

[14] Anna Kocurova et al. “MobWEL - Mobile Context-Aware Content-Centric Workflow Execution Language”. In: COLLA 2013, The Third International Conference on Advanced Collaborative Networks, Systems and Applications. July 21, 2013, pp. 61–70.

[15] Christian P Kunze. “DEMAC: A Distributed Environment for Mobility Aware Computing”. In: *Proc. 3rd Int. Conf. on Pervasive Computing*. Citeseer. 2005, pp. 115–121.

[16] Rüdiger Pryss, Julian Tiedeken, and Manfred Reichert. “Managing Processes on Mobile Devices: The MARPLE Approach”. In: CAiSE’10 Demos. Hammamet, Tunisia, June 2010.

[17] Gregory Hackmann et al. “Sliver: A BPEL Workflow Process Execution Engine for Mobile Devices”. In: *in: Proceedings of 4th International Conference on Service Oriented Computing (ICSOC)*. Springer Verlag, 2006, 503–508.

[18] Daniele Battista et al. “ROME4EU: A Web Service-Based Process-Aware System for Smart Devices”. In: *Service-Oriented Computing – ICSOC 2008*. Ed. by Athman Bouguettaya, Ingolf Krueger, and Tiziana Margaria. Lecture Notes in Computer Science 5364. Springer Berlin Heidelberg, 2008, pp. 726–727.

[19] Holger Schmidt and Franz J. Hauck. “SAMProc: Middleware for Self-adaptive Mobile Processes in Heterogeneous Ubiquitous Environments”. In: *Proceedings of the 4th On Middleware Doctoral Symposium*. MDS ’07. New York, NY, USA: ACM, 2007, 11:1–11:6.

[20] Rohan Sen, Gruia-Catalin Roman, and Christopher Gill. “CiAN: A Workflow Engine for MANETs”. In: *Coordination Models and Languages*. Ed. by Doug Lea and Gianluigi Zavattaro. Vol. 5052. Lecture Notes in Computer Science. Springer Berlin - Heidelberg, 2008, pp. 280–295.

[21] G. Tuysuz, B. Avenoglu, and P.E. Eren. “A Workflow-Based Mobile Guidance Framework for Managing Personal Activities”. In: *2013 Seventh International Conference on Next Generation Mobile Apps, Services and Technologies (NGMAST)*. 2013 Seventh International Conference on Next Generation Mobile Apps, Services and Technologies (NGMAST). 2013, pp. 13–18.

[22] *YAWL: Yet Another Workflow Language*.
http://www.yawlfoundation.org/.

[23] *MQ Telemetry Transport Protocol*.
http://www.mqtt.org/.

[24] Sonja Zaplata et al. “Flexible Execution of Distributed Business Processes based on Process Instance Migration”. In: *Journal of Systems Integration (JSI)* 1.3 (2010), pp. 3–16.

[25] Peter Khisa Wakholi and Weiqin Chen. “Workflow Partitioning for Offline Distributed Execution on Mobile Devices”. In: *Process Aware Mobile Systems. Applied to mobile-phone based data collection* (2012).

[26] Katharina Hahn and Heinz Schweppe. “Exploring Transactional Service Properties for Mobile Service Composition.” In: *MMS* 146 (2009), pp. 39–52.

[27] Rüdiger Pryss, Steffen Musiol, and Manfred Reichert. “Collaboration Support through Mobile Processes and Entailment Constraints”. In: *9th International Conference on Collaborative Computing: Networking, Applications and Worksharing (Collaboratecom)*. 2013, pp. 178–187.

[28] Gregory D. Abowd et al. “Towards a Better Understanding of Context and Context-Awareness”. In: *Handheld and Ubiquitous Computing*. Ed. by Hans-W. Gellersen. Lecture Notes in Computer Science 1707. Springer Berlin Heidelberg, 1999, pp. 304–307.

[29] Christian Becker and Frank Dürr. “On Location Models for Ubiquitous Computing”. In: *Personal Ubiquitous Comput.* 9.1 (2005), 20–31.

[30] Peter Fricke et al. “Towards Adjusting Mobile Devices to User’s Behaviour”. In: *Proceedings of the 2010 International Conference on Analysis of Social Media and Ubiquitous Data*. MSM’10/MUSE’10. Berlin, Heidelberg: Springer-Verlag, 2011, 99–118.

[31] Arthur M. Keller et al. “Zippering: Managing Intermittent Connectivity in DIANA”. In: *Mob. Netw. Appl.* 2.4 (Dec. 1997), 357–364.

[32] Samir Bellahsene and Leïla Kloul. “A New Markov-based Mobility Prediction Algorithm for Mobile Networks”. In: *Proceedings of the 7th European Performance Engineering Conference on Computer Performance engineering*. EPEW’10. Berlin, Heidelberg: Springer-Verlag, 2010, 37–50.

[33] Kun-Che Lu, Chen-Wei Hsu, and Don-Lin Yang. “A Novel Approach for Efficient and Effective Mining of Mobile User Behaviors”. In: *2010 4th International Conference on Multimedia and Ubiquitous Engineering (MUE)*. 2010, pp. 1–6.

[34] Seung-Cheol Lee et al. “Extracting Temporal Behavior Patterns of Mobile User”. In: *Fourth International Conference on Networked Computing and Advanced Information Management, 2008. NCM ’08*. Vol. 2. 2008, pp. 455–462.

[35] K. Gaaloul et al. “A Secure Task Delegation Model for Workflows”. In: *Second International Conference on Emerging Security Information, Systems and Technologies, 2008. SECURWARE ’08*. Second International Conference on Emerging

Security Information, Systems and Technologies, 2008. SECURWARE '08. 2008, pp. 10–15.

[36] Hermann Gehring and Andreas Gadatsch. *Eine Rahmenarchitektur für Workflow-Management-Systeme*. Fernuniv., 1999.

[37] R.S. Sandhu et al. "Role-based Access Control Models". In: *Computer* 29.2 (Feb. 1996), pp. 38–47.

[38] *Play Framework - Build Modern & Scalable Web Apps with Java and Scala*. *http://www.playframework.com/documentation/2.1.x/Home*.

[39] *The Jython Project*. *http://www.jython.org/*.

[40] *Activiti BPM Platform*. *http://www.activiti.org/*.

[41] *jBPM - JBoss Community*. *https://www.jboss.org/jbpm*.

[42] *AristaFlow - AristaFlow ® Next generation Business Process Management*. *http://www.aristaflow.com/*.

[43] Martin Odersky et al. *An overview of the Scala programming language (second edition)*. Tech. rep. École Polytechnique Fédérale de Lausanne (EPFL), 2006.

[44] *The Jython Book v1.0 Documentation*. *http://www.jython.org/jythonbook/en/1.0/*.

[45] Marco Berroth. "Konzeption und Entwurf einer Komponente für Organisationsmodelle". Diplom Thesis. University of Ulm, June 2005.

[46] *jBPM6 user Guide*. *http://docs.jboss.org/jbpm/v6.0.1/userguide*.

[47] *Activiti Userguide*. *http://www.activiti.org/userguide*.

[48] Peter Dadam et al. "From ADEPT to AristaFlow BPM Suite: A Research Vision has become Reality". In: *Proceedings Business Process Management (BPM'09) Workshops, 1st Int'l. Workshop on Empirical Research in Business Process Management (ER-BPM '09)*. LNBIP 43. Springer, Sept. 2009, pp. 529–531.

[49] *AristaFlow Project Page*.
http://www.aristaflow.de/.

[50] Manfred Reichert et al. "Enabling Poka-Yoke Workflows with the AristaFlow BPM Suite". In: *Proc. BPM'09 Demonstration Track*. CEUR Workshop Proceedings 489. 2009.

[51] Andreas Lanz et al. "Enabling Process Support for Advanced Applications with the AristaFlow BPM Suite". In: *Proc. of the Business Process Management 2010 Demonstration Track*. CEUR Workshop Proceedings 615. Sept. 2010.

Name: Steffen Musiol Matrikelnummer: 647792

Erklärung

Ich erkläre, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den . . . .

Steffen Musiol