Slovak University of Technology in Bratislava
Faculty of Informatics and Information Technologies

FIIT-5220-47937

Bc. Katarína Valaliková

# CONSISTENCY IN THE IDENTITY MANAGEMENT

Diploma thesis

Supervisor: Ing. Radovan Semančík, PhD.

2012, May

Slovak University of Technology in Bratislava
Faculty of Informatics and Information Technologies

FIIT-5220-47937

Bc. Katarína Valaliková

# CONSISTENCY IN THE IDENTITY MANAGEMENT

Diploma thesis

Degree course: Software Engineering

Field of study: 9.2.5 Software Engineering

Institute of Applied Informatics, Faculty of Informatics and Information Technologies

Supervisor: Ing. Radovan Semančík, PhD.

Education supervisor: Ing. Anna Považanová

2012, May

## Thanks,

to Ing. Radovan Semančík PhD., for his patience, the time which he invested to me and for presious advices.

# ANOTÁCIA

Slovenská technická univerzita v Bratislave
FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ
Študijný program: Softvérové inžinierstvo

Autor: Bc. Katarína Valaliková
Diplomová práca: Konzistencia údajov pri správe podnikových identít
Vedúci diplomovej práce: Ing. Radovan Semančík, PhD.

Pedagogický vedúci: Ing. Anna Považanová
máj, 2012

Systémy na správu podnikových identít sú integračné systémy slúžiace na automatizáciu procesov spojených s manažovaním používateľov. Pomocou systémov na správu podnikových identít sú riadené prístupy do rôznych koncových systémov, ktoré obsahujú citlivé dáta. Udržiavanie konzistencie medzi koncovými systémami sa preto javí ako dôležitá súčasť. Práca sa zaoberá problémami pri správe podnikových identít, ktoré môžu vyústiť do vzniku nekonzistencie medzi dátami a zároveň hľadaním mechanizmu, ktorý by dokázal eliminovať vznik takýchto nekonzistencií. Výsledkom práce je mechanizmus, ktorý je založený na troch princípoch, a to CAP teoréme, kompenzáciach a modeli relatívnych zmien. Navrhnutý mechanizmus pozostáva z dvoch častí, a to okamžitej reakcie na chyby a rekonciliácie. Rekonciliácia slúži na zistenie rozdielov medzi jednotlivými databázami a následne odstránením týchto rozdielov. Mechanizmus bol implementovaný do existujúceho systému sa správu podnikových identít s názvom midPoint a bude zahrnutý do nasledujúceho oficiálneho vydania (verzia 2.0). Testovanie mechanizmu bolo vykonávané takisto využitím systému midPoint.

Kľúčové slová: systémy na správu podnikových identít, konzistencia, dvojfázový odovzdávací protokol, ACID, Sagas, BASE, S3

## ANNOTATION

Slovak University of Technology Bratislava
FACULTY OF INFORMATICS AND INFORMATION TECHNOLOGIES
Degree Course: Software Engineering

Author: Bc. Katarína Valaliková
Diploma thesis: Consistency in the Identity Management
Supervisor: Ing. Radovan Semančík, PhD.

Education supervisor: Ing. Anna Považanová
2012, May

Identity management solutions are integrative solutions that automate processes associated with managing of users and theirs life-cycle. The identity management solutions provide accesses to different end resources, which can contain sensitive data. Consistency in the identity management is therefore an important issue. The thesis concerns on the issues related with the identity management which can lead to the inconsistencies. There is also tendency to find such mechanism which is able to minimize the risk of inconsistencies and if they ever happened it can reasonably react and bring the data to the consistent state. The result of the thesis is mechanism based on the tree essential concepts: CAP theorem, relative change model and compensations. Proposed mechanism consists of two parts: error handling and reconciliation. Reconciliation is used for finding the differencies among the databases and for eliminating these differencies. Mechanism was implemented and tested in one of the open-source solutions called midPoint and it will be included in the official release 2.0.

Key words: identity management, consistency, two-phase commit protocol, ACID, Sagas, BASE, S3

# Table of contents

# Introduction

Identity management systems are integrative solutions that manage identities and their access to various heterogeneous systems. The results of identity management solution can be thought of as  (loosely coupled) distributed systems. Many of the end resources managed by the identity management systems do not support transactions or other traditional consistency mechanisms therefore there is a real risk of inconsistencies. It is important to solve the “inconsistency problem” for many reasons. For example, the identity management solution interacts with various systems and information about user identities is stored in more than one database. Without any reliable consistency mechanisms the databases may diverge and it may not be clear which data record should be used.

Another reason why it is needed to solve the problem with inconsistencies may be security. The identity management solutions are security-sensitive systems because they manage accesses to other systems. Consistency of security policy is important to maintain good security level and also to be able to monitor overall security of an organization. For instance, potential attacker can intentionally cause inconsistency and escape the “security police”.

Many mechanisms for ensuring the consistency were proposed in the literature. There are many sources describing various mechanisms that can be used in the database systems. For instance, the two-phase commit protocol that uses lock and log mechanism is often used for distributed database systems, Sagas which are based on the sub-transactions with defined compensation mechanism and so on. However, these mechanisms are not suitable for identity management solutions. The aim of the thesis is to design practical mechanism which ensures the consistency of data in the typical identity management scenarios.

The thesis is organized as follows. Section 1 is devoted to analyze the current state of the art. It surveys the field of identity management and related technologies in general, the known mechanism dealing with the consistency issues and also the existing solutions of identity management. The section 2 defines the objectives of the thesis including specification of use cases. In the section 3 Solution design is described. There is provided detailed view on the mechanism design and also there are described the decisions made by designing the mechanism.

The section 4 Verification deals with the concrete implementation of the mechanism and provides the view on the way, how the mechanism was tested and evaluated. The section 5 Discussion is there to generalize the proposed mechanism and to discuss its strengths and weaknesses. The section 6 Conclusion summarizes the whole thesis and states the possibilities for the next work.

# 1 Problem Analysis

In this section the analysis of the current state will be provided. Sequentially, the intension will be made for the identity management and the common technologies used by it. Then the section will be concentrated to analyze the known mechanisms for ensuring the consistency in the databases and distributed systems. Finally there will be analyzed existing identity manager solutions to provide better view what they do for ensuring the consistency.

## *1.1 Identity management*

Identity management can be defined as a convergence of technologies and business processes [23]. It is integrative solution that usually consists of different systems and techniques [2]. Identity management solution cannot be the same for different companies because each company has its own requirements on the business processes and technologies so they need to have a specific instance of identity management. The main goal of identity management is to handle a lot of identities and their life-cycle including creation, usage, updating and revocation of the identity [2]. Identities have different roles and different permissions to access specified resources. There is a need to have different identities to work with the same system, or to have the same identity to work with different systems [23].

The definition of identity can be found in several resources [[18], [23]]. According to [18] identity is defined as attributes' set which can identify individual of any set of individuals. There is difference between "I" and "Me". While "I" refers to an instance accessible only by the individual self, "Me" refers to the social attributes defining human identity [18].

According to [23] identity can be defined as something (e.g., people, computers, applications) that is the same today as it was yesterday. Identity does not refer only to the person, but also to the computer or computer applications, that need to work with other systems. Today, a lot of enterprises use identity management to manage their employees. The need of identity management in the enterprises is obvious, for example:

- there is a lot of employees, that need to have access to the different resources,
- the roles of employees can change, and also their rights to the resource must be changed,
- the new employees are hired so they need to have also access to the some resources,
- some of them may be fired and should not have more access to the resources [23].

The main objectives of the Identity Management according to [23] are:

- "Enable a higher level of e-business by accelerating movement to a consistent set of identity management standards.

- Reduce the complexity of integrating business applications.
- Manage the flow of users entering, using, and leaving the organization.
- Support global approaches/schemas for certain categories of operational tasks.
- Respond to the pressure from the growing numbers of Web-based business applications that need more integration for activities such as single sign-on.”

### 1.1.1 Access management

Access management plays a crucial role in many systems. It is very important to secure resources especially in the systems with sensitive data such as identity management. First of all, authentication is needed to verify that the users are who they claim to be. After the user was authenticated, authorization is needed to permit user's access to specified resources. Next, main technologies used by identity management solutions are described.

One of the main technologies used in Identity Management solutions are Directory Services. Directory Services are used for storing users and their attributes in the tree structure. The common protocol used for accessing Directory Services is Lightweight Directory Access Protocol (LDAP) [[21], [23]] that is based on the X.500 protocol [24]. LDAP does not support transactions [24] or standards for access control [23], such details are specified individually for each LDAP deployment.

Other technology used in the identity management solution is Public Key Infrastructure [23]. Users are authenticated presenting their credentials, which is usually password or a public key certificate. The basic idea of authentication based on public key certificate is that the identity attributes are bound to the public key. Public key is unique for each user and according to this public key the messages can be encrypted and decrypted and only the user with the right public key is able to access them without additional sharing secrets between senders and recipients. The public key certificates are issued by certification authority and verification and life-cycle of public key certificates are controlled by Public Key Infrastructure [2].

Common technology used by the identity management solution is Single Sign-On systems. Authenticated user doesn't have to log in to different resources again and again. Single sign-on transaction is used to reuse authenticated user to access to different resources instead of repeated log-ins. This doesn't mean, that the single sign-on (SSO) systems unifies all user accounts for different resources, instead user can have more local accounts that are bound to the account used for authentication in SSO system (as shown in figure 1.). After the user was logged into the SSO system, other attributes including accounts characteristics are bound to the logged user. SSO system reduces the need of authentication for different systems using session [2].

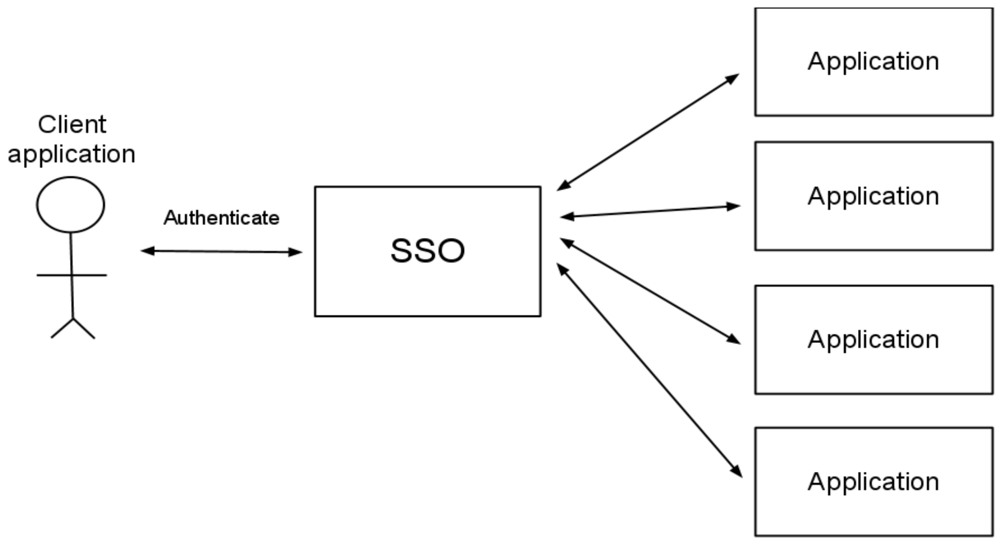


*Figure 1. Single sign-on architecture*

A variety of architectures can be used to implement single sign-on systems. Broker-based, Agent-based and Reverse proxy-based architecture are well-known [2]. Broker-based architecture is based on the central node, where the subjects are authenticated. After authentication they get the ticket, which can be used to access requested applications.

Agent-based architecture assumes that for each application server there is an agent, which plays a role of a translator between credentials and the authentication protocol used. Reverse proxy-based architecture as the name indicates, consist of the proxy, which filter identity credentials submitted by external subject, and if they are incorrect, the subject can be redirected to authentication server to get proper credentials. Proxy is usually located at the endpoint of the applications and the systems. Similar taxonomy is provided in [17].

The well-known broker-based protocol is Kerberos [11] that is used for client-server application. The authentication is performed using symmetric encryption. As the authors in the [2] mention, there is an authentication server which mediate authentication. After the client was authenticated, it gets a ticket which can be used for granting the rights to the various servers.

### 1.1.2 Role Based Access Control

Today applications need to manage access to secured resources. There are different ways to do that. One of them is Role Based Access Control (RBAC) [5]. RBAC defines access policies using roles. Users don't have individual permissions to secured resources, instead of that they are assigned to the corresponding role. The role has defined privileges which resources can access and which can't and it usually represents a job position [20].

RBAC is not a ready-made solution. It depends on a specific application and its requirements. The privileges, roles and constraints are defined according to the application and user can be assigned to more than one role. The benefit of RBAC is simple administration of users and their rights. For example, when employee changes his job position, new role is assigned to him, instead of setting all privileges related to the new job position [20]. Relationships between roles, users, objects (application to access) and constraints are shown in figure 2. The term transformation is used in the sense of applying defined constraints [5].

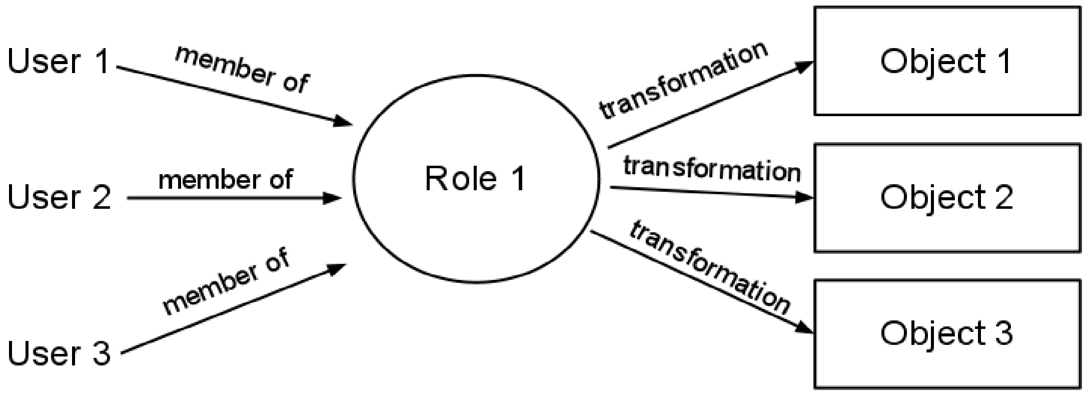


*Figure 2. Role relationships [5].*

Sophisticated RBAC solution is able to include role-role, permission-role and user-role relationships. This offers the ability to define role hierarchies, which means that the privileges can be inherited from the different roles. For example, if there is a role of project manager that inherits from the role of developer, the project manager would have privileges to project management resources and also to developer resources [20]. According to [20] RBAC supports three well-known security policies:

- Least privileges – means that to the roles are assigned only those privileges that are needed for the users in the role tasks.
- Separation of duties – in situation, when two mutually exclusive roles are needed to complete the task, using example from the paper [20] for instance requiring accountant clerk and account manager to participate in the issuing clerk.
- Data abstraction – means, that abstract permissions could be set instead of typical read, write and execute permissions used e.g. in the operating system.

Many applications use a concept of groups, but it cannot be mixed with the concept of roles. The group defines a set of users but not necessarily their privileges. Privileges for users belonging to specific group are defined separately for each user or resource, for example in the Unix, the privileges are defined for the each file. The role means a set of users and includes also their privileges [20].

It must be mentioned that RBAC is not a panacea to all access management issues. In some situations RBAC is not a sufficient solution. For example for situations that control operation sequence other access control mechanism should be used [20].

### 1.1.3 Identity provisioning

The main idea of identity provisioning is to automate processes for managing users. Provisioning systems manage changes that happened and replicate these changes to the different resources (figure 3.). For example, if a new employee is hired, there is a new record created about this person in Human Resource system. Provisioning system detects this new record, assigns the correct role to the user and according to this role creates accounts in the corresponding resources. Another example should be changed job position and so user's role was changed. Provisioning system detects this change and according to this new role creates corresponding accounts [21].

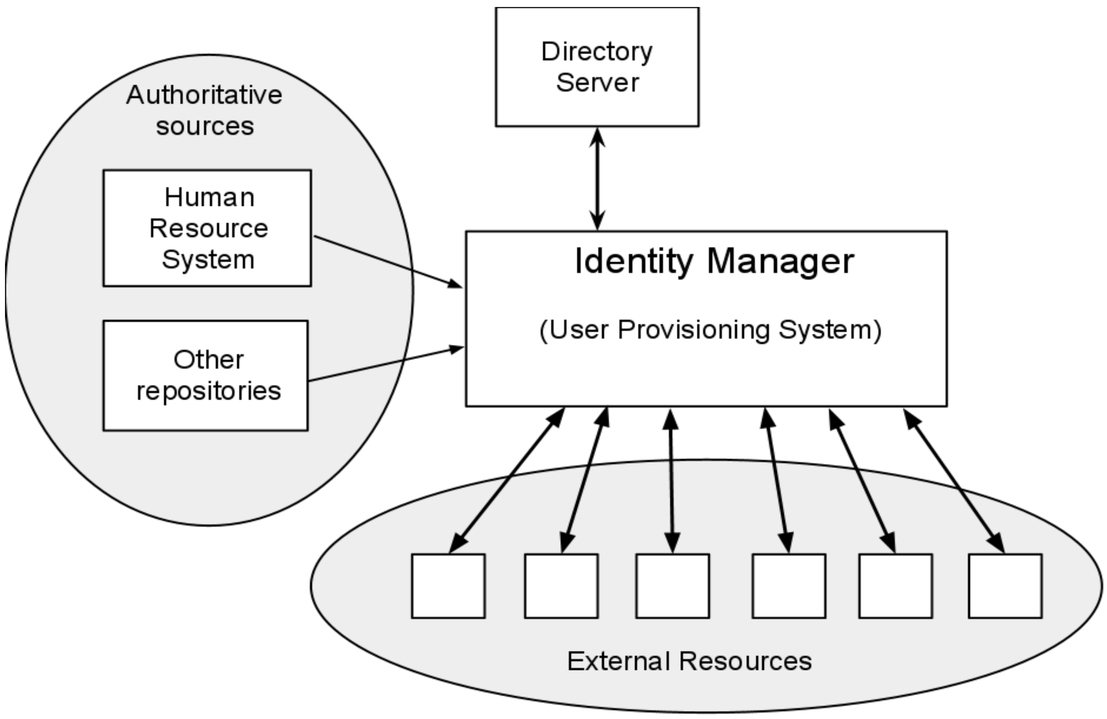


*Figure 3. Architecture of the Identity Management [22].*

Provisioning systems are well-equipped for managing user's records in the heterogeneous systems, where the single directory system fails. Reasons why directory system is insufficient are according to [21] following:

- There is not one source of information for a single user.
- In many cases there is a need for a local user database.
- Some services need to keep state.
- Different systems might have different meaning for role names and access control attributes.

There are two major problems related to provisioning system according to [21], risk of inconsistency and slow operation. Inconsistency may occur, when provisioning doesn't identify the change immediately or when some problem happened in the synchronization process. Problem with slow operation is associated with synchronization processes, which can take substantial amount of time.

Synchronization processes in the direction from information source to end systems are usually performed asynchronously and they are able to immediately react to the change. But sometimes an interaction with user is needed which may take significant amount of time. In the reverse direction, from end systems to provisioning systems, the synchronization processes are not usually performed asynchronously and they may take several hours or days.

## *1.2 Consistency*

This section describes various mechanisms that are used to provide consistency, such as ACID transactions, Sagas, BASE, Simple Storage Service. Each mechanism is described in more detail to explain the main principle. CAP theorem according to which in distributed systems there cannot be guarantee of consistency, availability and partition tolerance at the same time is also mentioned.

### 1.2.1 ACID

Consistency in the database systems is guaranteed by transactions. According to [8] transaction can be described as transformation from one state to another while maintaining atomicity, durability and consistency. Authors in [8] equate transactions to the contract, where at least two parties are required, and the contract is successfully concluded only if all parties agree. Transaction management is responsible for transactions to decide whether the transaction would be applied (operation was successfully) in the database or rolled back (operation failed).

Transactions are described using ACID properties, which stands for A – atomicity, C – consistency, I – isolation, D – durability. Atomicity means that all operations appear as one that is either successfully completed or aborted. If an operation is aborted the initial state is preserved and no one can detect that anything happened. Isolation means that if the transactions run concurrently, each transaction appears as if there were no other transactions at the same time. Durability means, that if the transaction completed successfully and the changes were written into the database, such changes are permanent. And consistency means, that the consistency constraints must be preserved when new data are added [25].

Transactions in the distributed systems are realized using two-phase commit protocol. Two-phase commit protocol (2PC) is responsible to coordinate all of the participants in the transaction and to decide if the transaction will be committed or rolled back. Before all transaction are committed or aborted, the data in the database are locked to enforce concurrency control [26]. As the name two-phase indicates, there are two phases called voting phase and decision phase. In the voting phase, the participants are able to abort the transaction and in the decision phase the participants decide whether the transaction will be committed or aborted with maintaining atomicity of the transaction [1].

### 1.2.2 Sagas

Long living transactions are introduced in [6], transactions that can execute hours or days. Applications that use such long transactions cannot use standard lock and log mechanism, mechanism used in database systems using ACID transactions.

The paper [6] describes an approach where a long transaction called *saga* is divided into shorter sub-transactions. Such sub-transactions can be mixed-up with any others transactions. Each sub-transaction has its own compensation mechanism. If transaction failed, compensation mechanism is called to undo this transaction. It necessarily doesn't mean that compensation mechanism return database to the same state as before transaction run because it is able to run another transaction using this data.

If *saga* is interrupted by failure, there are two mechanisms to recover from this error, backward recovery and forward recovery. Backward recovery means that each sub-transaction of *saga* has its own compensation mechanism that is invoked to restore previous state. Running sub-transaction is rolled back and for completed sub-transactions is called compensation mechanism in reverse order [6].

In the case of forward recovery, the *saga* execution component needs to have reliable copy of code and save points. If the save points are defined at the beginning of each sub-transaction, the process is called pure forward recovery. The running sub-transaction is aborted and *saga* is restarted at the point where this aborted sub-transaction has started. But, if there are no save points at the beginning of each sub-transaction, it is called backward/forward recovery. In this case executing sub-transaction is aborted and other sub-transactions compensation mechanism is called to restore the state to the save point and then *saga* is restarted [6].

Critique of this approach is provided in [9]. Author explains that *sagas* fails to cover all cases where compensation is needed, using an example of e-procurement scenario. For example, author mentions, that in some cases there is a need to write null compensator. Such a compensator acts as if the compensation ended successfully and enclosing scopes have no idea if the activity of undo was performed correctly.

Compensation mechanism might be not correct also in other cases, e.g. if there are concurrent activities. In such case the compensation mechanism has access to the state captured by the original activity and also to the stored state in database, but there is no knowledge about current state of running activity. The example of this case is demonstrated in [9] using order cancellation case and the current state of the order delivery. The appropriate rollback mechanism is affected by whether the invoice was sent to the customer or not.

According to this paper the compensation mechanism for the transaction is not satisfactory in all cases. Sometimes there is a need for doing more than just call the compensator to undo the operations. However, even if the compensator succeeds, it is not guaranteed to restore the original state. This can result in relaxed consistency requirements but ensuring eventual consistency (eventually the data would be consistent) [9].

### 1.2.3 CAP Theorem

Services in the computer network usually try to achieve three distinct quantities:

- guarantee strong consistency – such as database systems using ACID transactions,
- achieve high availability – services should be as accessible as the network where they run,
- provide fault-tolerance – services should act as expected, even if some node crash.

According to CAP theorem [7] it is impossible to guarantee consistency, availability and partition tolerance at the same time in distributed systems. The CAP theorem explains that it is possible to guarantee only two of them. The paper [7] provides a proof of the theorem using asynchronous network model.

Availability means that every request must have appropriate response apart how long the process is running. Even if some crash occurs or the node is down, the process must be eventually terminated. Consistency should be described using atomic objects. It means that the operations must act atomic, as if they were executed at a single instant. In the [7] authors explain consistency on the concept of distributed shared memory. Requests in the distributed shared memory are processed one a time, and outwardly they look as if they were performed on a single node. The concepts of availability and consistency are introduced with respect to the partition tolerance which allows losing some messages send from one node to another.

At first, authors in the [7] describe different combinations of two guarantees (consistency – partition tolerance, consistency – availability, availability – partition tolerance) in the asynchronous network model and they mention situations in which each of combination should be used. Then the notion about weaken consistency in the partially synchronous model is introduced.

It depends on the system characteristics when to prioritize consistency instead of availability. If availability is chosen it means that the consistency would be weaken. Then if partition fails, some messages should be lost and therefore inconsistencies may occur. If the operations are commutative it can be easy to restore the consistency. Many commutative operations have non-commutative exception, what means that they are late or incorrect. It brings the possibility for more operations using the concept of non-commutative exceptions to be considered as commutative and it can simplify the eventual consistency. In the paper [4] author thinks about this exceptions also as compensations which can be used to restore consistency.

At the end of the paper [4] author mentions that in fact most real wide-area systems use concept of compensation mechanism to restore consistency. In such as systems the audit trails and recovery are preferred to prevention and author suggests expanding and formalizing of the role of compensation in the complex systems.

### 1.2.4 Basically Available, Soft state, Eventually consistent

BASE means Basically Available, Soft state and Eventually consistent system [19]. This approach is opposite to ACID, where each operation must result in consistent state of database. BASE allows temporary inconsistence of database systems. It allows partial failures without total system failure.

Whereas BASE provides availability it results according to CAP theorem to relax consistency. But the consistency issues are very important for the success of the application so there must be guarantee that the database would be eventually in the consistent state. Author in the [19] suggests to establish persistence message queue and to divide the database tables across functional groups, such as table for user, transactions, messages, etc. The problem with the persistence messages is that to avoid the usage of two-phase commit protocol (2PC), there must be guarantee that the backing persistence is on the some resource as the database. Otherwise, if the user host is included, there is a 2PC situation.

Idempotent operations with persistent message queue can be considered as a solution to allow for partial failures without using two-phase commit protocol. Idempotent operations are the operations that can be executed one time or multiple times with the same result. However, update operations are mostly not idempotent and if there is no guarantee that the order of the update operations would be preserved, the system can get into the wrong state [19].

Storing the main properties of transaction in the database table should be one of the possible solutions which are able to guarantee eventual consistency. This properties could consist of transaction ID describing the update operation and the user ID to which the transaction was applied. Messages describing what should be done with user properties are stored in the message queue. Each message is peeked from the queue and is processed. If the operation was successful, the message is removed from the message queue otherwise it is re-tried [19].

Two transactions may be considered, one for the database operations and one for the message queue with assumptions that database transaction are committed before message transactions. Such an algorithm supports partial failures without using two-phase commit protocol [19].

### 1.2.5 Simple Storage Service

Amazon provided Simple Storage Service [3] several years ago. Simple Storage Service (S3) serves as distributed storage mostly for multimedia data. Users can use S3 to store their data for a small payment. S3 uses both, REST and SOAP interface and try to reach scalability and high availability by replicating stored data and try to provide consistency as much as possible. S3 together with Simple Query System (SQS) and Elastic Computing Cloud (EC2) belongs to Amazon Web Services.

Simple Query System (SQS) is used to manage (virtually) infinite number of queues which are able to have (virtually) infinite capacity. Queues consist of the messages which can be sent using both REST-based and HTTP-based interface. Each message in the queue has a unique id, according to which the message can be read, locked and deleted form the queue. Messages may contain any byte stream, no pre-defined schema is required. Clients are never blocked by other clients or system failure, the request by the clients can be initiated at any time [3].

According to [3] if user updates some stored data it takes some time to view updates also by another users. It means that data can be partially inconsistent so S3 should only guarantee that eventually data became consistent. This is also known as eventual consistency. The replicas can be updated at any time by any user, clients are not blocked by other clients or system failures. If more than one user updates the replica, replicas are merged using reconciliation process on the “last win” basis. It brings persistence guarantee which means that updated replicas can be changed or undone only with new update request. The reconciliation process also guarantees full read and write availability even when a data center fails. In this case, the last updated replica in other data center is used to merge all replicas in the same state.

The protocol suggested in the [3] consists of two steps. In the first step, the log records about each update committed as a part of transaction is generated and then in the second step the checkpointing applying log records to the stored pages on S3 is used. The first step should be performed at a constant time, what follows from assuming that clients are never blocked and that the SQS are virtually always available. The second step, checkpointing includes the synchronization process. Even using synchronization process, the clients are never blocked by other client or the system failure. The synchronization process can be performed asynchronously, whit no impact on the execution of the client application.

The protocol is also resistant to the failures. In the case that the client fails during commit, the log records are resent. Then log records are applied again and it can result to applying the same records twice. However, this is not a problem, because log records are idempotent, which means that applying them one or more times result in the same state. But if the client crashes during commit, it is also possible that some log records may be lost or the client cannot come back. This case violates the atomicity, because some log records after the client crash are not applied [3].

## 1.3 Consistency in Identity Manager

In this section existing solutions of identity management are described. It was chosen open-source projects such as OpenIAM, OpenIDM, MidPoint and OpenPTK. The focus is on the provisioning part, which provide synchronization and reconciliation processes. This processes are used to manage flow of changes and to propagate this changes to desired resources. If the changes are not detected immediately or some crash occurred, it can result to inconsistencies.

### 1.3.1 OpenIAM

OpenIAM is an open-source identity and access management project. This solution is based on the Service Oriented Architecture (SOA) and it combines both SOA and Identity standards. The heart of the OpenIAM is an Enterprise Service Bus (ESB) through which services are exposed. The ESB brings portability, which means that services can be consumed using different technologies. OpenIAM supports all common use cases of identity management, for example creates new user, provisions user into systems which he needs to access, revokes user's access when he left the company [12]. Architecture of the OpenIAM solution is presented in the figure 4.

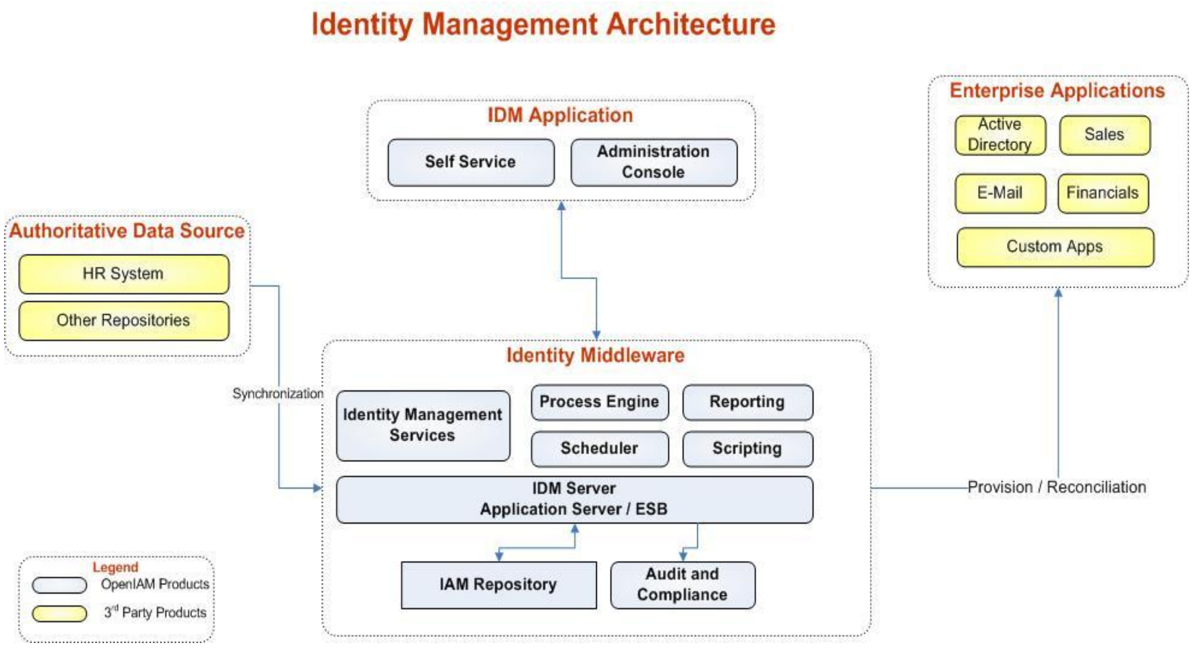


*Figure 4. Architecture of the OpenIAM solution [12].*

OpenIAM as well as other identity management solutions needs to protect resources and manage access to them. For that reason, authentication is needed. According to project wiki page [12] OpenIAM provides authentication based on Password and Security Tokens, but there is an opportunity to extend authentication service to provide other forms of authentication using Login modules. Password Authentication might be used with different repositories, such as LDAP repository, Active Directory (AD) or relational database.

The provisioning engine of the OpenIAM supports synchronization of the resources managed by OpenIAM. Synchronization processes are initiated in the case of events when user is created, modified, terminated or the password is changed or reset. It is also possible to define additional events that are relevant for the organization. There are two ways of detecting an event found on the project wiki page [12]:

- Event Based – where application specific module is created to detect changes in the real time.
- Reconciliation based – used when event based approach is not possible, e.g. some applications do not allow to detect the changes of users. Reconciliation allows detecting changes through plugin or polling. In the former case (through plugin), the plugin is integrated to the application provider and if the change occur, the plugin is called. The latter case, detecting changes through polling requires the configuration file for each source system. This configuration file consist of the elements describing system, query which extracts the information, OpenIAM record to compare with, frequency of calling process etc.

There are Synchronization Routing interfaces [12]:

- Routing Web Service,
- SPML 2 Router – XML Profile and
- SPML 2 Router – DSML 2 Profile.

Routing services receive messages which describe the change occurred. Each authoritative source must have defined the validation policy and the modification policy which needs to be performed by routing interface. These policies can be configured through the OpenIAM web console interface. The validation policy is used to check the content of the received messages and to check, that the authoritative source provides only those attributes which it owns. Modification policy is used in the cases, when some additional information to the message or the transformation to the suitable form is needed. This additional information may be derived from the known information [12].

Routing services perform the validation and modification policies and then the decision about which provisioning connector to use may be made. There are several ways to do this decision as the project wiki page [12] mentions. In the case of the new user a Job Role based or a Policy based decision can be used. If the Job Role based decision is made, the user is provisioned according to his role describing the needed accesses. Policy based decision have defined policies according to which the user is provisioned. These policies usually contain job codes, departments, etc.

The OpenIAM project has a very poor documentation which is also inconsistent. The architecture overview is brief and there are no details about the design. The technical details are included in the developer documentation section, which may be a little bit confusing. The consistency issues are not mentioned on the project wiki page [12]. The only remark to a potential consistency problem was found in the OpenIAM issue tracking system Jira [13]. It also seems that the community is not sufficiently strong because only few names repeatedly appear in the issue tracking system and also on the project wiki page. The project has roadmap for the next year, but there are only releases planned and no details about features supported in each next release.

### 1.3.2 OpenIDM

OpenIDM is an open-source identity management solution. The OpenIDM team is trying to bring lightweight, developer friendly, modular and flexible solution of identity management. The architecture of the OpenIDM is shown in the figure 5.

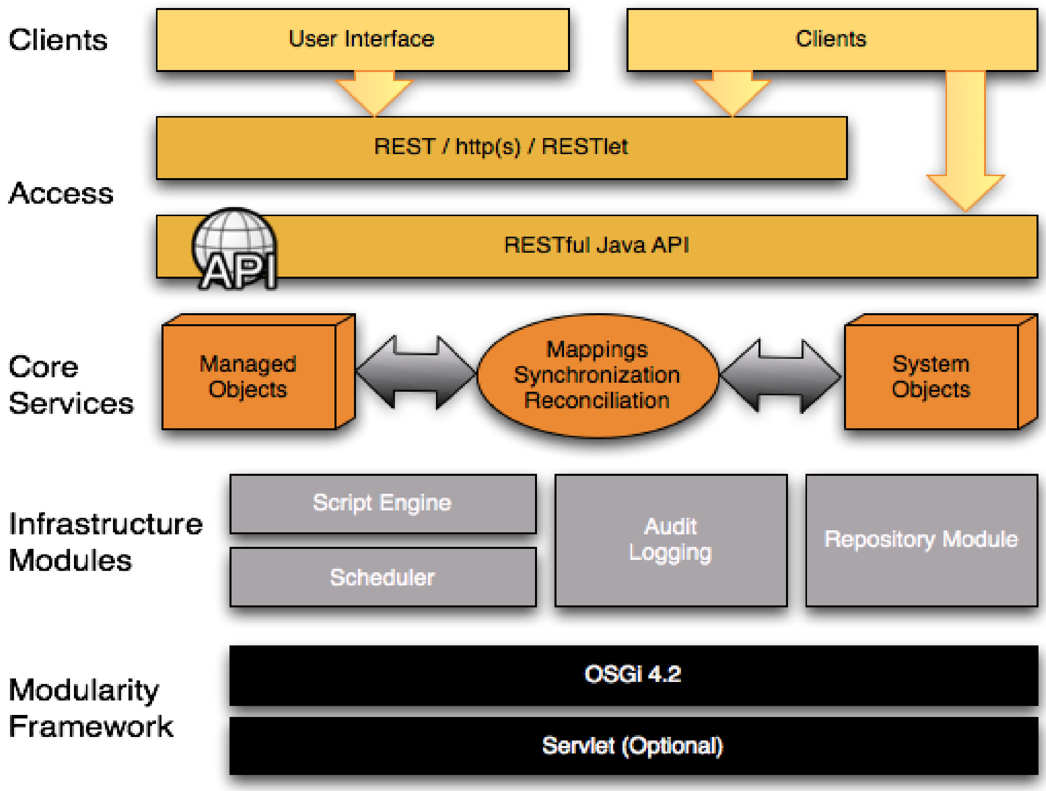


*Figure 5: OpenIDM architecture [15].*

OpenIDM solution, as shown in figure 5 is divided into five layers, Modularity Framework, Infrastructure Modules, Core Services, Access and Clients. Modularity Framework, as the name implies, is supposed to provide modularity to the OpenIDM and to reduce complexity of the solution using OSGi. RESTful HTTP access to the managed objects and services can be done with the Servlet technology, which is optional [15].

The aim of the Infrastructure Modules layer is to provide the functionality needed for the core services. It consists of:

- Scheduler that is mostly used by synchronization and reconciliation.
- Script Engine that provides triggers and plugin points for OpenIDM.

- Audit Logging which is used to log operations on the internal managed objects and external system objects, and also by the reconciliation as the basis for the reporting.
- Repository, standing for persistence layer. There can be NoSQL, relational databases, LDAP or flat files repositories.

The Core Services layer is the heart of the OpenIDM. There are definitions of objects and features that OpenIDM can do. Managed objects describe identity-related objects managed by OpenIDM using JSON-based data structures. These objects are stored in the OpenIDM repository. Representation of the external resources objects is done through the System objects. Policies how should be managed objects transformed to the system objects and opposite are defined through the mapping. These mapping policies are used also by synchronization and reconciliation [15].

Access layer provides the public API and the user interface for communicating with the OpenIDM repository and its functions. Using REST API there is a possibility to make your own user interface [15].

OpenIDM supports synchronization and also reconciliation. These two processes are used to detecting changes between system objects and managed objects and according to the defined policies process these changes. However, the documentation does not contain any mention about the inconsistencies which should occur when something went wrong by these processes. In addition, the OpenIDM documentation is very confusing, finding out the desired information take a lot of time. The structure of the OpenIDM wiki page is not logical. There is a lack of information about the particular components of OpenIDM, e.g provisioning part.

### 1.3.3 OpenPTK

OpenPTK is an open-source User Provisioning Toolkit. It was developed under the Sun Microsystems community program and now it seems that the Oracle took over the project. OpenPTK serves as the tool for developing user management systems. The sample applications are also provided to make the usage of the OpenPTK easier. Web developers and Java developers can integrate custom applications with the provisioning system using OpenPTK [16].

The aim of the OpenPTK is to provide the toolkit, which can make the development of user management systems easier. Because of each enterprise has different requirements and developers may have various experiences with various technologies, OpenPTK provides various client interfaces. Project is built on the concept of a three tier architecture allowing developers to focus on the business logic and hiding the work with the underlying repositories. The three tiers according to project wiki page [16] are:

- Server Tier – provides RESTful Web Service (supporting JSON, XML).
- Framework Tier – integrate Server Tier and Service Tier, support authentication, authorization, representations, monitoring, logging, debugging, request/response

etc.

- Client Tier – interfaces that extend the RESTful Web Services. There are several development options to choose from.

Service Tier provides the back-end user data repository [16]. The architecture of the OpenPTK solution is shown in the figure 6.

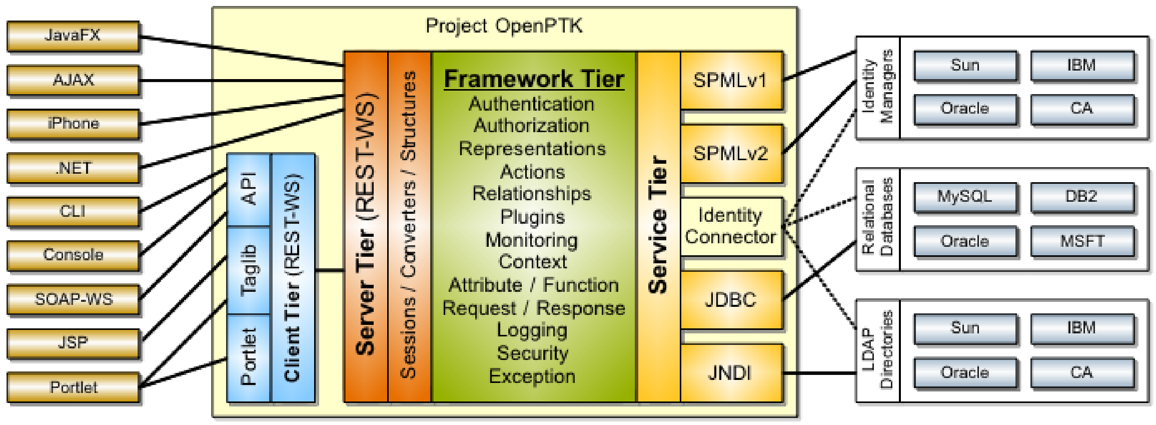


*Figure 6. OpenPTK architecture [16].*

The documentation to the OpenPTK project is very brief. It is missing an overview about architecture or mechanisms used in the project. The project wiki page [16] is confusing and finding out desired information may take a lot of time. Documentation has the form of tutorial explaining how to use OpenPTK rather than the architectural guide which describes what was done and why it was done in this way. Consistency issues were not founded on the project wiki page [16]. The OpenPTK is developed under Oracle supervision so it may imply a community with more members.

### 1.3.4 MidPoint

Architecture of the midPoint is designed to bring the modularity to the system. The main idea is to bring a set of components that should be composed together according to the needs of the certain company. Components can be likened to the Lego bricks and they can be built together as needed. This architecture solution allows easily add new component, remove component or replace existing component with the custom solution according to company needs [10].

Subsystems of the midPoint listed on the project wiki page [10] are User Interface Subsystem, Bussiness Logic Subsystem, IDM Model Subsystem, Provisioning Subsystem, Persistence Subsystem, Utility Subsystem and Infrastructure Subsystem (Figure 7.). Each of them carries out different function [10]:

- User Interface Susbsystem provides interaction with user implementing end user interface or administration interface.
- Bussiness Logic Interface carries out the business processes that are implemented

using BPEL. Such business processes might be handling of special cases, provisioning processes, handling of exotic situations etc.

- IDM Model Subsystem contains implementation of Role Based Access Control, Rule Based Access Control, which are helpful by automatically creating accounts on the target systems (according to user role) or detection of account attributes (according to policy rules).
- Provisioning Subsystem provides management of accounts, accounts attributes, groups, roles, etc.
- Persistence Subsystem which main function is persist the data.
- Utility Subsystem provides stateful services used by many components covering for instance configuration, monitoring, management, rule processing, policy evaluation, etc.
- Infrastructure Subsystem contains stateless component, such as logging, tracing, spring and similar libraries.

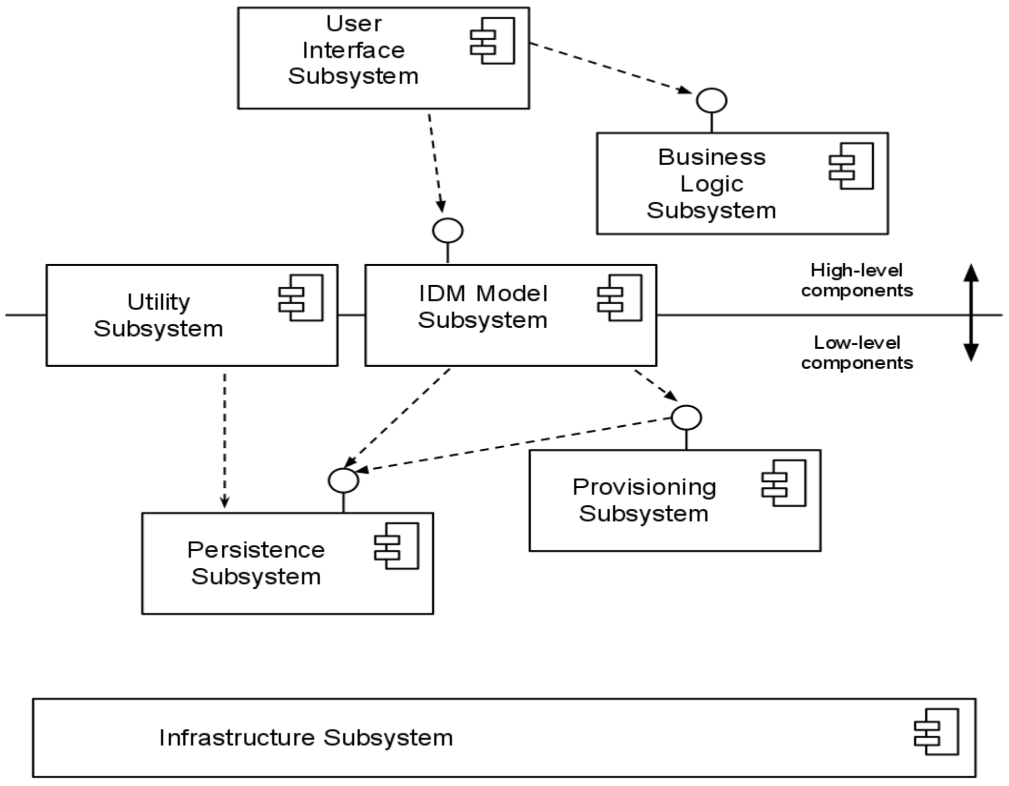


*Figure 7. Architecture of the midPoint solution [10].*

Provisioning subsystem of midPoint is responsible for managing accounts, groups, roles, entitlements and other objects related to identity management. The aim of provisioning part is to connect midPoint with other managed resources including directory servers, human resource servers, mainframes, etc. It provides communication among these external Resources and manage changes which are able to occur in Resources. Changes that occurred are propagated by synchronization processes [10].

Synchronization process synchronizes data to be as up-to-date and complete as possible. It means that the synchronization process propagates changes from midPoint to external resources and vice versa. The data in the midPoint itself and in the external Resources may not be the same, but the intention is that they are not contradictory. Synchronization process is supposed to be near-real-time process that should quickly respond to the changes that occured. But, in some situation the synchronization process is not sufficient. For example, the external Resource might be down so the changes might not arrive or some changes may be missed so they are not propagated to the desired resources. In such situation, the reconciliation is needed [10].

Reconciliation process manages changes by comparing the absolute state of the resources and IDM which might bring better reliability than the synchronization process. Disadvantage of reconciliation is that it is a long task so it cannot run very often. The effort of the midPoint is to use the same interfaces for synchronization and reconciliation processes so they could be unified. The changes are then processed in the same way apart of the process which detect them [10].

MidPoint has a rich documentation, covering architecture and design of the solution in the depth. The division into the individual sections is logical and less confusing as compared to similar documentation of the OpenIAM project. The project wiki page [10] also mentions the consistency issues and it is still open problem. Community involved in the project seems to be stronger than the team of OpenIAM solution.

# 2 Objectives

The goal of the thesis is to find an appropriate way to solve the consistency issues in identity management systems. The identity management system must be able to recover from unexpected errors and continue to work without limiting the users. It is unacceptable to allow identity management to be in the inconsistent state for a long time because this could result to the malfunction of the system. It is also important to solve the inconsistencies because of security of the system. Identity management system usually holds secure data about the users, it manages the access to the various external systems and if the inconsistencies will not be solved this should harm the security of the system.

In this section there are first described the identified situations by which the inconsistencies should occur. Situations are divided into the categories according to their nature. Next, the intension is made for specifying the goals of the thesis. After the goals are specified, use cases are introduced to cover the consistency issues of the relevant parts of the solution.

## 2.1 Identified situations

Identity management systems provide automation of the processes related to the users and their life-cycle in the company, from hiring new employee through changing his position to firing employees. Each of employees usually has multiple accounts in the various systems to be able to perform his work properly. Therefore there are a lot of external resources which need to communicate with the identity management systems. External resources contain information about the employees and their access rights. One employee should have accounts in the different resources and may also have more than one account in the same resource.

Accounts are created in different ways, e.g. using central identity management system, by synchronization of changes on external resources, or by adding the user to the role which defines that an account should be created etc. Unexpected situations and errors may happen during the user management processes, e.g. the account may not be created, exceptions may be thrown, etc. Ultimately, this may lead to the inconsistency of the record. Many situations by which inconsistencies may occur exist. According to the way they originate, they should be divided into the following categories:

- Resource failures – this group describes failures that happened on the external resource by propagating changes that was made by end user using identity manager (e.g. add account on the external resource, modify account on the external resource, etc.).
- Synchronization failures – describes failures that happened by synchronization. Changes on the external resource was detected and propagated to other external

resources and also to the identity manager but some crash occurred.

- Dependencies – describes inconsistencies that should happened by creating account that may have dependencies to other accounts (e.g. Account in the application is depended on the account in the operation system. The account in the application should be created only if there is an account in the operation system, etc.).
- Groups – describes failures that happened when some operation with the group was made (e.g. Creation of account and adding it to the group are in LDAP two different operations.).
- Role Assignment – describes inconsistencies that occurred while working with roles (e.g. User has assigned role which describes that four different accounts should be created, but only two of them are created successfully, the role is in the inconsistent state.).

Defined categories contain many kinds of operations. These operations are described in detail in the Appendix A – Requirements Analysis and Specification in the part A.1 Situations. For the further thesis purposes the decision is to consider only the one of the identified groups of problems.

## 2.2 Goals of the Thesis

The goal of the thesis is to find such mechanism which will be able to minimize the formation of inconsistencies and if they ever happen, this mechanism will be trying to resolve them and bring the system back to the consistent state. This mechanism will be designed with respect to the existing identity management solutions and technologies. The known mechanisms used by existing solutions and the mechanisms used by traditional transactions system will be also supposed.

The next aim of the thesis is to design and implement such mechanism and in this way provide the practical proof to the solution. The solution for ensuring the consistency will concentrate on the fact that most of the external resources connected to the identity manager are not transactional and so they mostly don't support transaction operations. Therefore there is need to find new mechanism similar to that used by traditional transactions systems described earlier in the section 1.2 Consistency.

The goal of the thesis can be also formalized to supporting transactions in the non-transactional environment. It means using benefits and ideas of transactions and expanding them to be able to use them also by the non-transactional systems.

## 2.3 Use Case Model

Categories of situations that lead to inconsistencies were identified in the section 2.1 Identified situations and they are detailed described in the Appendix A – Requirements

Analysis and Specification (part A.2 Use cases). From the presented categories the resource failures were chosen for a more detailed analysis. The resource failures category describes the inconsistencies which can occur by propagating the changes from the identity manager to the external resource.

Situations belonging to this category will be used to define what is needed to be solved. The use cases identified from these situations which will be implemented in the system are:

- Add account extended with the failures of
    - schema violation,
    - object already exist,
    - generic error and
    - error in the communication with the connector.
- Delete account extended with the failures of
    - object not found on the resource,
    - generic error and
    - error in the communication with the connector.
- Modify user or accounts attributes with the failures of
    - objects not found,
    - schema violation,
    - generic error and
    - error in the communication with the connector.

All these identified use cases are described in detail in the Appendix A – Requirements Analysis and Specification.

# 3 Solution Design

The content of this section will concentrate to design the solution to the presented use cases in the previous section. The solution proposed in this thesis will not be created on a green field. It will be rather based on an existing identity management platform. First, the substantiation for chosen starting platform will be provided, comparing the pros and cons of analyzed platforms in the section 1.3 Consistency in Identity Manager. The chosen platform will be described in more details in Appendix B – MidPoint Product Description to provide better view on its architecture and also the state of the platform development before the work on this thesis began.

The summary of the analyzed mechanisms follows in the section 1.2 Consistency and the substantiation of chosen mechanism will be made. Next chapter will be devoted to design the mechanism for ensuring the consistency with respect to the chosen identity manager solution. In this chapter there will be introduced some concepts and ideas how should these mechanism behave in various situations. After description of concrete situations and the behavior of the consistency mechanism in these situations, the summary for the consistency mechanism will be provided.

The diagrams used in this section are based on the UML concepts, but they are not strictly following the UML specification. Primarily, they are used to describe the ideas of the designed mechanisms that can be understand by engineers, therefore formal correctness of the diagrams is relaxed in favor of understandability. In the sequence diagram used in the section, the mechanisms and the parts proposed as a part of this thesis are color-coded to clearly distinguish them from the parts that existed before or were developed independently by other developers.

## 3.1 Choosing the Best Solution

The scope of this thesis includes a practical proof of the theoretical concepts discussed in it. Therefore an appropriate platform is needed as a basis to implement the consistency mechanisms. It is obvious that the scope of this work cannot include creating a complete identity management system from the ground up. The idea of creating a mock (incomplete implementation) identity management system just for this thesis also does not seem appropriate. Such minimal implementation will be still very difficult to implement and the quality of a demonstration using such an incomplete system would be questionable. Therefore it was decided that use of an existing identity management system and extending it with consistency mechanisms seems to be the best approach.

Numerous open-source identity management systems were considered as was described in section 1.3 Consistency in Identity Manager. MidPoint identity management project was chosen as the basis for this work. Comparing midPoint with other mentioned open-source identity management solution, the midPoint seems to be the best choice. It has better documentation than others with the clear description of all its parts. Also principles and goals are described more in depth than by other mentioned solutions.

The comparison of wiki pages and theirs structure also brings the reason why to choose the midPoint solution. By the others solution there is confusing structure, it takes a lot of time to find desired information. The wiki pages except midPoint wiki page are mostly poor in information. They seem to be not maintained regularly. Some important details, e.g about provisioning or synchronization are missing on these pages. The aim of this thesis practical part is to add consistency guarantee to the existing solution, but it could be considerably hard to do this without the good knowledge of the existing solution.

With respect to the consistency mechanism it is a significant advantage for this thesis to choose an architecture that allows for adding consistency mechanisms without breaking the existing design and philosophy of the system. By studying existing solutions other than midPoint, there was no point discussing the consistency issues. It should mean that these solutions do not take care about the data consistency or they do this without telling how and this is not suitable for the thesis purposes.

At midPoint wiki page [10] there is a mention about following the weak consistency model which means that there are no guarantees of the data consistency all the time. However, the idea is to have the mechanism which will be able to minimize the risk of inconsistencies and if they nevertheless happened, this mechanism will be able to solve the problem reasonably. Consistency mechanism has been not proposed yet in the midPoint and it is still open issue. For the thesis this may be the right place where to start with the practical proof. This fact also brings another reason why to choose midPoint solution.

However, all of the mentioned solutions are open-source, and it means that it is able to modify or add implementation to the existing one. The choice was influenced also by the fact that I am a member of the midPoint’s development team so I have practical experiences with this product. This brings same advantages for me, such as the architecture is well-known for me or that I can influence the development. This fact contributed to the other arguments specified above to choose midPoint as a basis for the practical part of this thesis.

## 3.2 midPoint Solution in Detail

It is very important to have detailed information about the solution in which the mechanism will be placed. Therefore in the Appendix B – MidPoint product Description there is provided the deeper view into the midPoint’s basic principles and supported features in the release 1.9. At a time when this part of the thesis was written, the release 1.9 was the latest release of the midpoint.

For the thesis purposes, the provisioning part of the midpoint was chosen. This part seems to be the best for placing the consistency mechanism, since it is the part responsible to communicate with other external resources. The mechanism there can immediately react to the errors occurred on the external resources. The provisioning part and also its current state in the release 1.9 are detailed described also in the  Appendix B  – MidPoint Product Description.

For the sake of completeness the table 1 defines some of the common terms used in the midPoint project and also in the Identity Management field. Because of that they will be used also in the next text. The terminology is kept in the main text of the thesis because it is needed for understanding the right meaning of the further text. In the further text, these terms will be written using *italics* style.

*Table 1: Commonly used terms in midPoint*

| **Term** | **Definition** |
|---|---|
| Resource | target or end system, which is supposed to be connected with the midPoint |
| Resource Objects | objects like accounts, groups, roles, entitlements or similar objects that exist on the resource. The structure of these objects is based on the resource where they exist. |
| Shadows | objects like accounts, groups, roles, entitlements or similar objects that are stored in the midPoint repository. These objects are local copies of any resource objects, they represent resource object in the midPoint but they have another form as Resource Objects. |
| Objects | All objects that can in midPoint exist, e.g. user, account, group, role etc. |
| Properties | Individual properties related to the specific object, e.g. user has properties: name, givenName, fullName etc. |
| End user | User who is working with the midPoint. It is not the object, but the human being. |
| Account | It is used to describe the account which exist on the external resource.It is a complete account with all attributes. |
| Account shadow | Shadow of the account. It is used to describe account which is stored in the local midPoint’s repository. This account is not complete. There are only mandatory fields and attributes contains only identifiers (from the resource account). |
| ICF | Identity Connector Framework – connectors used for communication with the external resources. |
| UCF | Unified Connector framework – the extension of the ICF that should fix some known ICF issues. |

## 3.3 Choosing the Mechanism

Each of the situations described in the previous section should be solved to avoid of inconsistencies in the system. These situations were divided into five categories to which the solution should be found. This thesis will not deal with all of them because of the current identity management system state described in the  Appendix B  – MidPoint Product Description. The chosen category which will be discussed to the depth is resource failures.

Proposed solution will follow the model of the eventual consistency which means that midPoint would be not guarantee that data will be consistent all the time. Instead, the temporary inconsistencies will be allowed and the attention will be made for the mechanism which solves the inconsistencies and eventually brings the data to the consistent state. This model of eventual consistency may be compared to the midPoint weak consistency model.

There exists several reasons why to use weak consistency (or eventual consistency) instead of strong consistency in software like midPoint is. They results from fact, that midPoint is an integrative solution that integrates a lot of target systems and in this way the loosely coupled distributed system is built. For such system, it is required to guarantee high availability and so you can read and write to the system all the time. Every request to the system must have appropriate response even if some crash occur (e.g. one of the node is down). It doesn't matter if the operation was successful, but it must be terminated and the result returned to the user. Even, if some message sent from one node to another is lost, the system must continue to operate.

According to CAP theorem described in section  1.2.3  CAP Theorem it is impossible to simultaneously guarantee availability, consistency and partition-tolerance in the distributed systems. Because the midPoint is trying to guarantee availability and partition-tolerance, the consistency is weakened. In addition, many of the target systems usually don't support traditional ACID transactions and it also impacts the consistency. It is influenced by the fact that identity management provides long running transactions which should run several hours or days. The standard lock and log mechanisms ensuring the strong consistency used by traditional ACID transactions will be therefore not suitable to use for such long running transactions.

In the section  1.2.2  Sagas there was also introduced compensation transactions for long running transactions called Sagas. In this way, one transaction is divided into the sub-transactions and each sub-transactions has defined its own compensation mechanism. If the sub-transaction failed, the compensation mechanism is invoked to restore the changes to the state before the sub-transaction run. If all sub-transactions are successful, the transaction is also successful. But there was a critique to this approach that discusses the problem with the coverage of the problematic cases and explains difficulty for creating compensation mechanism for some situations.

In addition, in some cases even if the compensation mechanism is able to be defined and it seems to be successful, it cannot be with certainty said that it really reflects the previous state. This problem is also possible to be solved introducing the eventual consistency. For those facts, the eventual consistency seems to be the best choice for the thesis and it will be considered also by further design.

## *3.4 Resource Failures*

Resource failures are related to execution of some operation in identity manager and propagating these changes to the external *resources* which should generate some error. Some of the errors are processable and they should be treated. At the beginning it is important to divide the errors into the processable and unprocessable errors and then design the solution for them.

Under the processable errors we consider errors which can harm the consistency constraints and we know to write the compensation to eliminate the issues. Under unprocessable errors we consider errors which do not harm the consistency constraints and also we do not know to write the compensation for them. While the errors are depended on the executed operation, they will be discussed later.

It is obvious that there is no sense to treat the unprocessable errors. Such errors will be just propagated to the *end user* to tell him what went wrong and the solution is left to the *end user*. In the case of processable errors, the appropriate reaction should be designed. The reaction is conditional upon the operation as well as the error which invokes the reaction. The most common operations by which such processable errors should occur are:

- add account,
- delete account,
- modify account attributes and
- modify users attributes.

### 3.4.1 Add Resource Account

Operation add resource account is responsible for creating new *account* for existing user on the specified *resource* and also for creating *account shadow* in the local repository. The *account attributes* should be added manually by the *end user*, or the *resource* should have defined policies – outbound expressions, according to which the attributes are automatically generated. By adding *account* to the external *resource* four different exception listed in table 2 should occur.

Exceptions described in the table 2 are divided into processable and unprocessable. The Schema violation exception is assigned as unprocessable. It means that the mechanism will not be interesting in these exceptions. This exception might mean that the *end user* doesn't fill required attribute. In this case, it has no sense to process this exception and try to re-add the account because the result will be always the same – it fails because of missing attribute. Therefore the schema exception is thrown to upper layer and also to the *end user* with the clear message for him. Then, it depends on the *end user*, what to do next, if he will try to add missing attribute and re-add the account or he will do nothing.

*Table 2: Exceptions by the add resource account operation.*

| **Exception type** | **Description** | **Processable** |
|---|---|---|
| Schema violation | Exceptions binding to the schema, e.g. some required attribute is missing. | NO |
| Generic error | Various exceptions depended on the connector type, e.g. it can be some SQL exception. | NO |
| Communication error | Exceptions with communication with the external resource, e.g. timeout exception, connection refused exceptions, etc. | YES |
| Object already exist | Exceptions that object we are trying to add already exist on the external resource. | YES |

Under the Generic errors various exceptions should be hidden. But this exception is rare and it should not result to the inconsistencies. Therefore no attention will be made for such errors. They will be simple thrown to the upper layer and also to the end user with the clear message what went wrong. The next operation is left on the *end user*.

More interesting exception for the thesis practical proof is thrown by communication problems. This can be the timeout exception or connection refused exception etc. Such exceptions are processable and they should be solved. If we get the timeout exception we don't know what was done in fact. The timeout exception should be thrown during the response phase and so the *account* could be created and it really exists on the *resource*. However, the midPoint got the timeout exception and it doesn't know about created *account*. On the other hand, other exceptions should be thrown and the *account* couldn't be created.

In such situation, it will be probably the best to store the requested account to the repository as *shadow*. This shadow will contain all attributes that the *account* should have and midPoint can try again later to add this *account*. It results to the creation of mechanism which will be able to scan midPoint's repository for such unsuccessfully handled *shadows* and find those which should be processed again later. This mechanism will be the reconciliation process described later.

Finally, there may be a situation when the *account* already exists on the external *resource*. This situation is more complex than the previous. The reasons, why the *account* already exists may be different. For example the midPoint was down and the administrator manually adds *account* to the external *resource*. Since the midPoint was down, the change of the new *account* wasn't detected and so it wasn't propagated to the midPoint. Besides how was the consistency problem formed, it must be solved but the question is how it should be done.

First, the *shadow* for arrived *account* is created in midPoint repository and then it must be find out if the *account* on the external *resource* is legal what belongs to the *model* (understand one of the midPoint component) responsibilities. The *account* and the obtained facts will be therefore sent to the *model* to find out what to do next. If the *model* declares the *account* as legal, than it is possible that it is the *account* we want to add for the user.

Ownership of *account* can be identified by the correlation and confirmation rules adjusted in the *resource* definition. Correlation rule describes policies according to which the owner of the *account* is found. Actually, the result of the correlation rule can contain more than one *account* owner and then the confirmation is needed. The confirmation rule is used to specify more concrete rules to find the concrete user.

Evaluation of these rules is also the responsibility of the *model* component so it will be ask *model* to make the decision. If the *model* declares *account* on the *resource* as the one that should be created, it is only needed to link this *account* to the existing user. Otherwise, the new identifier for the *account* must be generated and then try to add it again.

The situation when the *account* is illegal is quite different. If the *model* declares *account* as illegal, the *account shadow* and the *account* existing on the resource must be deleted and new *account* added. In the provisioning can be this situation obtained by looking up the *resource* for this *account* and if no match is detected it just means that the previous existing *account* was illegal, but it was removed and so correct *account* may be added. Object already exist situation is shown in figure 8.

### 3.4.2 Delete Resource Account

When the delete account is called, the result should be that the *account shadow* is removed from local midPoint's repository and *account* is removed from external resource. The operation delete account should be called in many ways, e.g. the user was deleted and also all his accounts should be deleted or the specific account should be deleted etc. By deleting *account* from external resource, there might be some issues obtained. The issues leads to generate the errors listed in the table 3.

*Table 3: Errors by delete resource account operation.*

| **Exception type** | **Description** | **Processable** |
|---|---|---|
| Generic error | Various exceptions depended on the connector type, e.g. it can be some SQL exception. | NO |
| Communication error | Exceptions with communication with the external resource, e.g. timeout, connection refused, etc. | YES |
| Object not found | Object we are trying to delete in not on the resource. | YES |

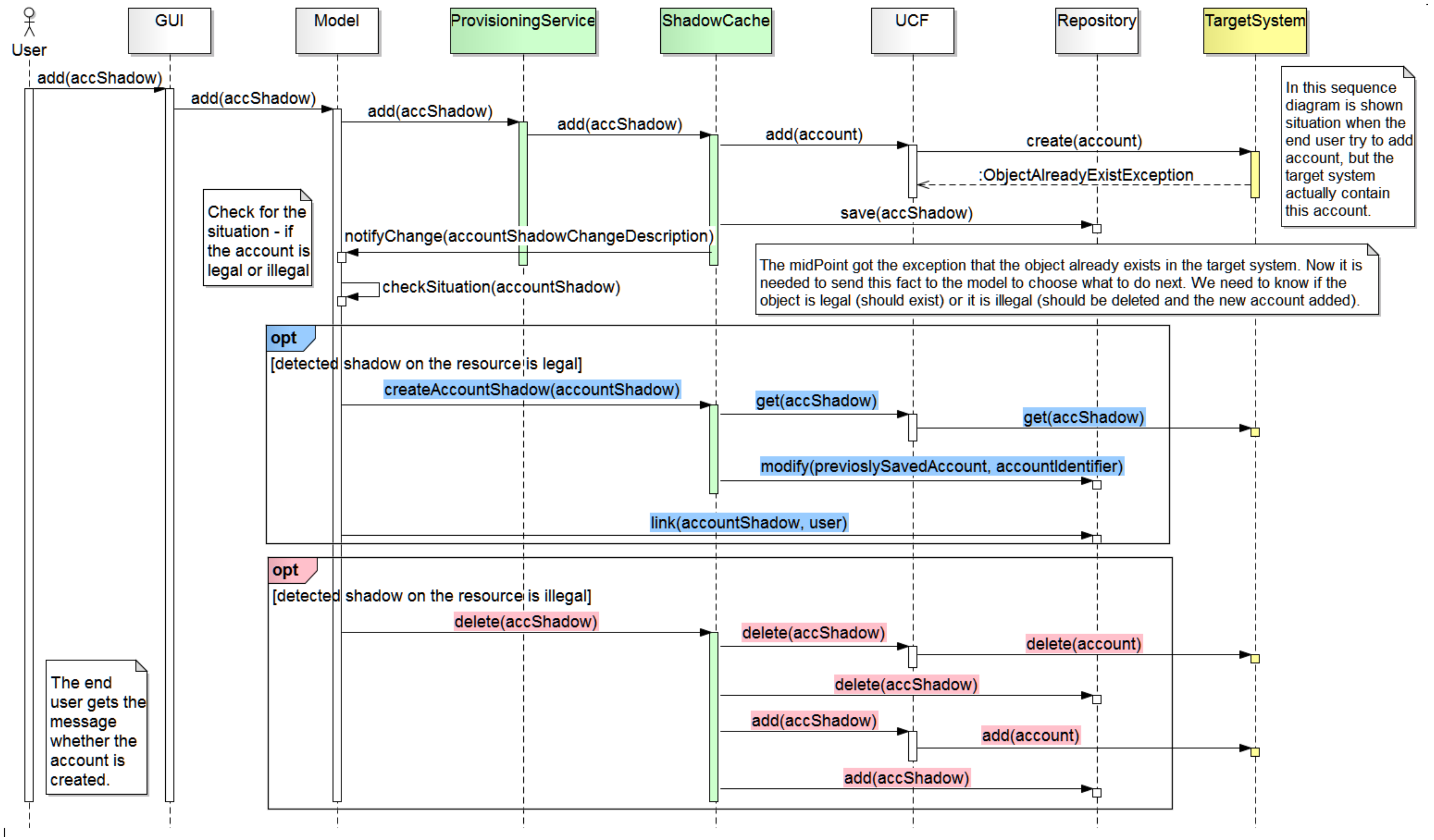

User
GUI
Model
ProvisioningService
ShadowCache
UCF
Repository
TargetSystem
add(accShadow)
add(accShadow)
add(accShadow)
add(accShadow)
add(account)
create(account)
:ObjectAlreadyExistException
save(accShadow)
notifyChange(accountShadowChangeDescription)
checkSituation(accountShadow)
Check for the situation - if the account is legal or illegal
In this sequence diagram is shown situation when the end user try to add account, but the target system actually contain this account.
The midPoint got the exception that the object already exists in the target system. Now it is needed to send this fact to the model to choose what to do next. We need to know if the object is legal (should exist) or it is illegal (should be deleted and the new account added).
opt
[detected shadow on the resource is legal]
createAccountShadow(accountShadow)
get(accShadow)
get(accShadow)
modify(previoslySavedAccount, accountIdentifier)
link(accountShadow, user)
opt
[detected shadow on the resource is illegal]
delete(accShadow)
delete(accShadow)
delete(account)
delete(accShadow)
add(accShadow)
add(account)
add(accShadow)
The end user gets the message whether the account is created.

*Figure 8: Sequence diagram for situation add resource account - object already exists.*

As in the previous scenario of add account, also by the delete account action there must be specified which errors will be processed and which won't. The generic error is probably something what cannot be influenced by the system and it means that it has no sense to handle this error. It can happen rarely and its results should not lead to the inconsistencies because nothing is done on the target system.

The situation by communication error while deleting *account* is quite similar than was by the adding account operation. It is also needed to know about this operation for the future. Since, it is not clear if the *account* was actually deleted from the *resource* (e.g. the timeout exception was thrown during response), it is needed to store the information to the midPoint's repository that this *account* should be deleted and sign it as the "dead" *account*. Later, calling reconciliation process the *account* is tried to be deleted again from *resource* and *account shadow* from midPoint's repository. This situation is shown in figure 9.

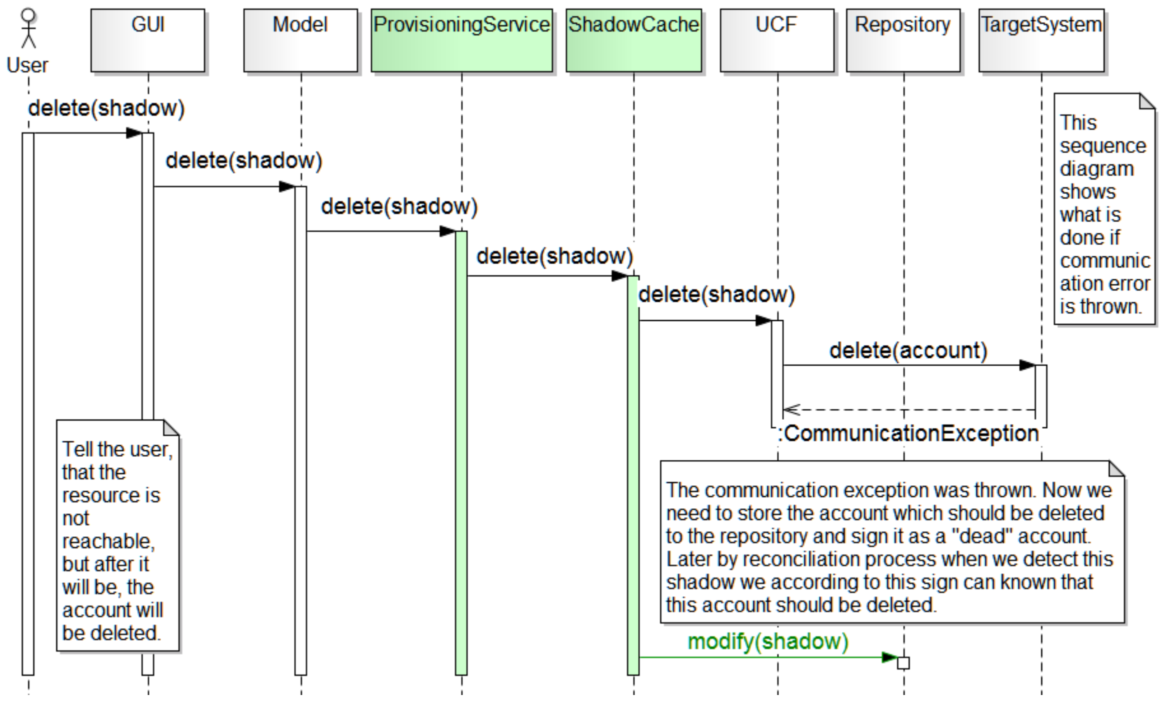

User
GUI
Model
ProvisioningService
ShadowCache
UCF
Repository
TargetSystem
delete(shadow)
delete(shadow)
delete(shadow)
delete(shadow)
delete(shadow)
delete(account)
:CommunicationException
modify(shadow)
This sequence diagram shows what is done if communication error is thrown.
Tell the user, that the resource is not reachable, but after it will be, the account will be deleted.
The communication exception was thrown. Now we need to store the account which should be deleted to the repository and sign it as a "dead" account. Later by reconciliation process when we detect this shadow we according to this sign can known that this account should be deleted.

*Figure 9: Sequence diagram for situation delete resource account – communication problem.*

By deleting *account* it can be also thrown exception that the required object was not found on the external *resource*. It means, that the user tries to delete absent *account* from the target system, but *account shadow* to this *account* seems to be still stored in the midPoint's repository and the *user object* is linked with this *account*. The reasons why there is the *account shadow* stored in the repository but not on the *resource* can be several, e.g. the *account* was deleted from the *resource*, but the synchronization process didn't detect this change so the change wasn't propagated to the midPoint.

In this situation seems to be the best to lookup the midPoint's repository if the *account shadow* still exist here and if the match is found, the *account shadow* will be deleted from midPoint's repository. If the match was not found the error probably is that the *user object* is still linked with the absent *account*, so we need to remove this link between the user and the account. This situation is probably good to solve immediately when the error is thrown so the end user should get the message if the account was deleted successfully or the operation failed.

### 3.4.3 Modify Resource Account

Modify resource account operation is supposed to modify the *accounts* attributes on the *resource*. The change of account attributes might be conditioned either by change of user's object or directly by changing the account's attributes. As well as by the previous use cases also by modifying account's attributes there might happened some unexpected situations which avoid to correct termination of the modify operation. These situations are described in the table 4.

*Table 4: Errors by modify resource account operation.*

| **Exception type** | **Description** | **Processable** |
|---|---|---|
| Schema violation | Exceptions binding to the schema, e.g. some required attribute is missing. | NO |
| Generic error | Various exceptions depended on the connector type, e.g. it can be some SQL exception. | NO |
| Communication error | Exceptions with communication with the external resource, e.g. timeout exception, connection refused exceptions, etc. | YES |
| Object already exist | Exceptions, that object we are trying to add already exist on the external resource. | YES |

The schema violation errors and generic errors are also categorized as unprocessable like by the previous scenarios of add and delete resource accounts. By these errors it is not responsibility of the midPoint to solve the problem, because they are coupled with the interaction with *end user*. The solution is to throw the error to the upper layer and lets the *end user* to choose what will be done next.

The communication error is similar as it was by add and delete resource accounts. In the situation when the external *resource* throws the communication error, for the midPoint it is not known what was done in fact. We also need to store the state of the required changes of *account* to be able to find out later what fails. By the modify account we therefore need to store the description of changes and apply this description later by the reconciliation process.

If we get the error that the *account* cannot be found on the external *resource* it can be pretty interesting. The *end user* wants to modify attributes of *account* that is no more on the external resource. First, we need to find out if the *account* should exist on the resource or if it shouldn't. But in the provisioning where this mechanism will be applied we do not have such a possibility. This is the responsibility of the *model* component that can find out if the user should have the *account* according to the user's assignments. These assignments are stored with the *user object* and they describe which accounts, groups or roles should user have.

The *model* component according to the user's assignments decides that the user either should have the *account* or shouldn't. If the decision is made for the positive example (user should has this *account* on the *resource* but it was deleted for some reason), model component send the provisioning the message to create the *account* on the external *resource*. After the *account* is created on the external *resource*, the changes should be applied.

If the model component decides, that the *account* actually should not exist on the *resource*, the changes description is discarded and the *account shadow* is removed from the midPoint’s repository. The *end user* gets the message that it is not possible to modify the *account* because of the detected inconsistencies of the system. The additional information to this message will be that the *shadow* was deleted from the local repository because it should not exist. The modify resource account when the object not found exception is thrown is shown in figure 10.

### A.1.1 Reconciliation Process

Now, it is needed to specify the policies for reconciliation process according to which the unsuccessful created *accounts* will be picked from repository. This brings the idea that storing only *account shadow* attributes is not enough. If there is not additional information to such *account shadow*, it is quite impossible to declare it that it was unsuccessful and to find it in the repository for the later usage. Therefore it is needed to expand the *account shadow* with some additional information which provides the detailed view about what and why went wrong.

There are three different operations which were supposed to create inconsistencies. Each operation work with different *object types* and therefore they must be considered by expanding the account. In the situation when the account should be added but the operation fails, it is probably good to have stored:

- all the *account* attributes (not only identifiers, but also other attributes describing the account),
- operation result which will give the status of the operation and also the error which occurs (it also provides the hierarchical results of all called operations),
- some information about the operation and the type of error which occurs,
- number of the attempts made for re-try of operation.

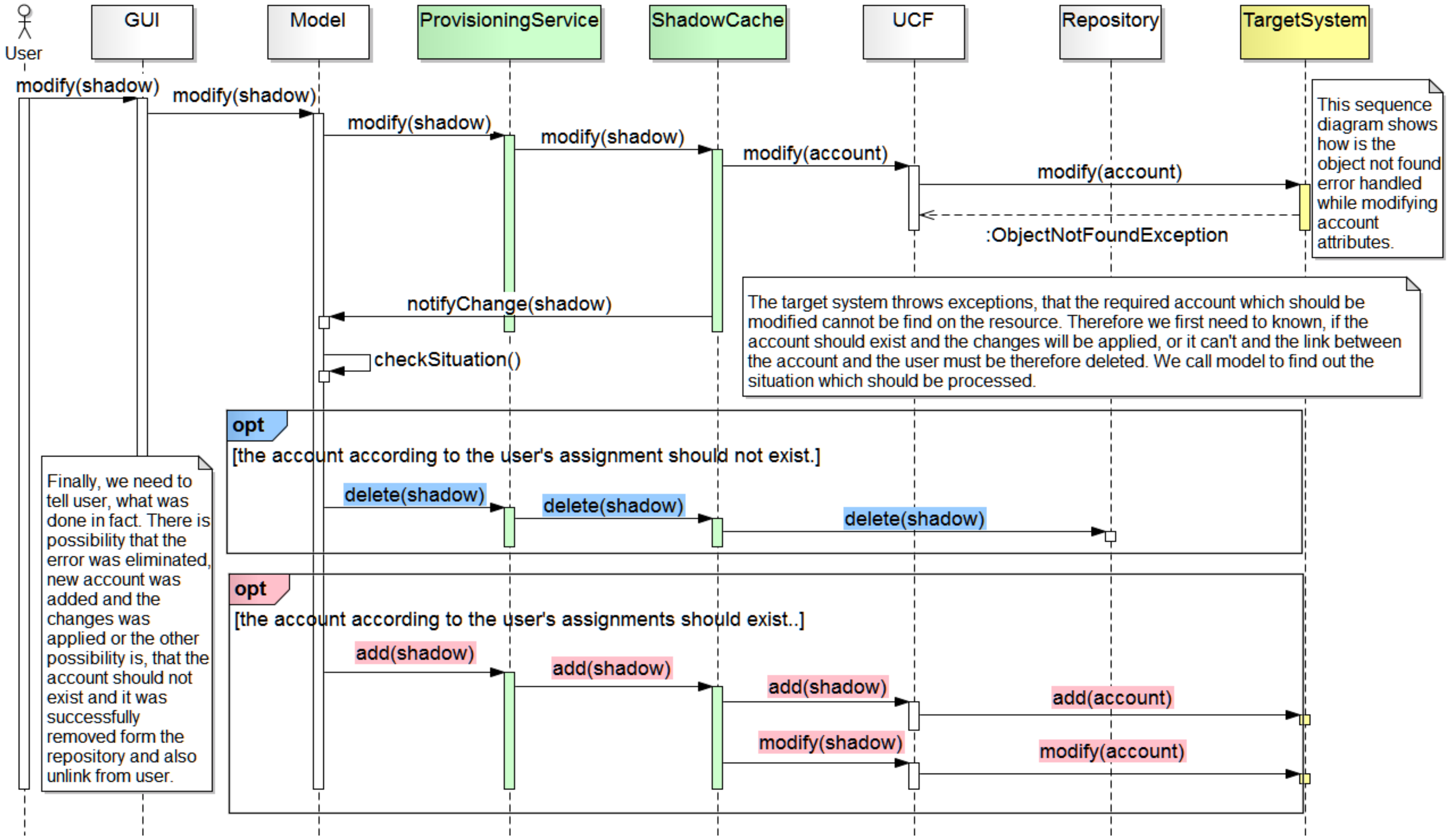

User
GUI
Model
ProvisioningService
ShadowCache
UCF
Repository
TargetSystem
modify(shadow)
modify(shadow)
modify(shadow)
modify(shadow)
modify(account)
modify(account)
:ObjectNotFoundException
This sequence diagram shows how is the object not found error handled while modifying account attributes.
notifyChange(shadow)
checkSituation()
The target system throws exceptions, that the required account which should be modified cannot be find on the resource. Therefore we first need to known, if the account should exist and the changes will be applied, or it can't and the link between the account and the user must be therefore deleted. We call model to find out the situation which should be processed.
opt
[the account according to the user's assignment should not exist.]
delete(shadow)
delete(shadow)
delete(shadow)
opt
[the account according to the user's assignments should exist..]
add(shadow)
add(shadow)
add(shadow)
add(account)
modify(shadow)
modify(account)
Finally, we need to tell user, what was done in fact. There is possibility that the error was eliminated, new account was added and the changes was applied or the other possibility is, that the account should not exist and it was successfully removed form the repository and also unlink from user.

*Figure 10 Sequence diagram for situation modify resource account - object not found error.*

By the modifying account there is probably good to have stored:

- modifications description of account (it means relative changes to the account which should be applied),
- operation result which will give the status of the operation and also the error which occurs,
- some information about the operation and the type of error which occurs,
- number of the attempts made for re-try of operation.

For deletion of account it is probably good to have stored:

- account which should be deleted, but hasn't been deleted yet (it means that the account shadow will be not deleted from the midPoint’s repository, but rather it will be signed as a “dead” account),
- operation result which will give the status of the operation and also the error which occurs,
- some information about the operation and the type of error which occurs,
- number of the attempts made for re-try of operation.

All the proposed expansion of *account shadow* object above might be summarized in the following one, storing:

- all the account attributes when operation fails,
- description of changed properties,
- operation result which will give the status of the operation and also the error which occurs,
- some information about the operation and the type of error which occurs,
- the number of the attempts made for re-try of operation.

Now, the *account shadow* in the repository contains also information about operations and theirs results. The reconciliation process will be used to scan the repository for the failed accounts and to try to run the failed operation again. The scanning will be done with filter on the failed accounts but the important role will also have the number of the attempts which was made to retry the unsuccessful operation. The number of attempts is important to prevent for repeating the re-try operation endlessly.

If the account is found according to the specified filter, the system triggers the failed operation with the account again. The information about which operation went wrong and what was the reason of crash can be found in the account's properties stored in the repository. If the account should be added, add account operation giving the account (with all properties stored temporary in the repository) will be called. If the account should be deleted, delete operation should be called and if the account should be modified, the modify account will be called. Input for modify account will be the change description stored with the account in the midPoint’s repository.

The reconciliation process ends either successfully or it also can fail. It will be implemented in the way to not limit the end user for his activity. After defined number of attempts it will not be interesting in the solving the inconsistencies and it will try to restore the state before the failed operation occurs. The idea of reconciliation process is shown in figure 11.

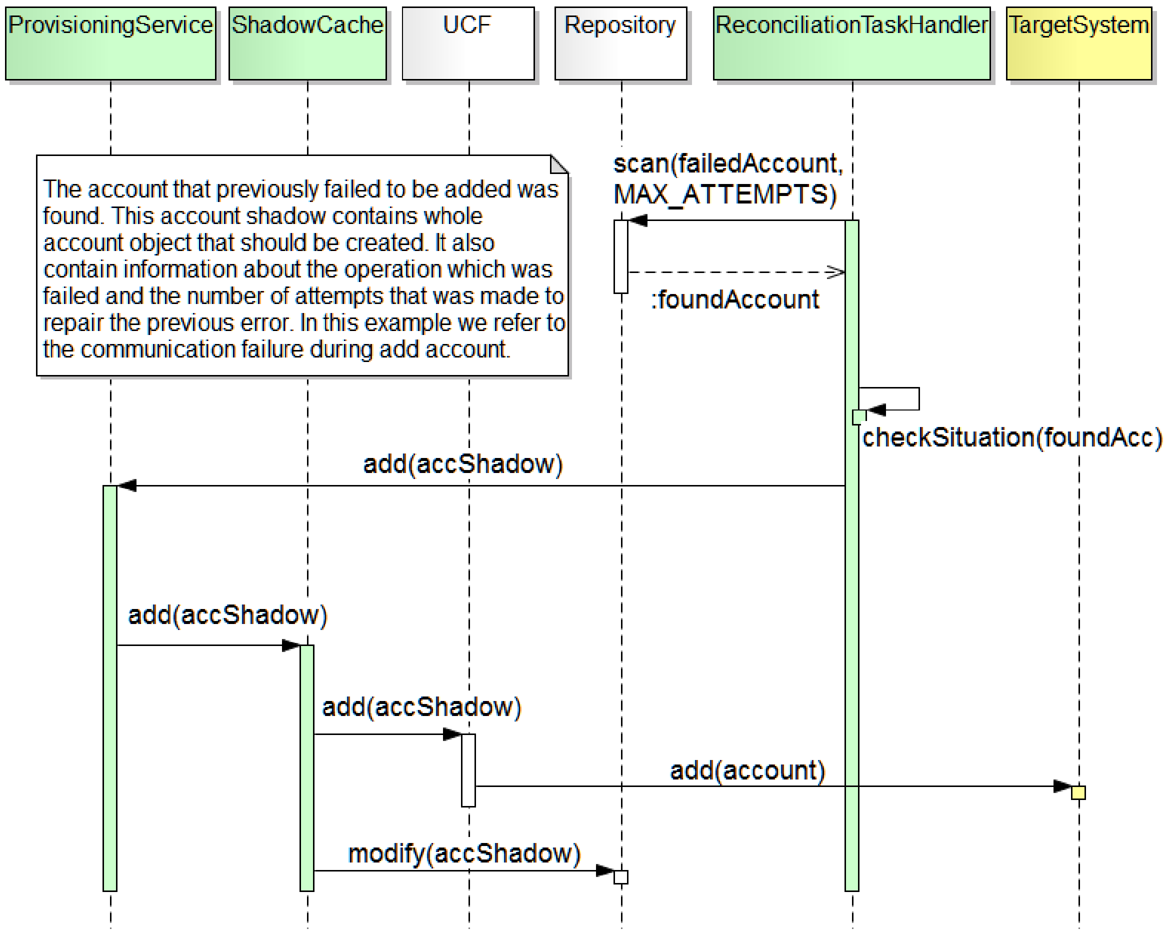

ProvisioningService
ShadowCache
UCF
Repository
ReconciliationTaskHandler
TargetSystem
The account that previously failed to be added was found. This account shadow contains whole account object that should be created. It also contain information about the operation which was failed and the number of attempts that was made to repair the previous error. In this example we refer to the communication failure during add account.
scan(failedAccount, MAX_ATTEMPTS)
:foundAccount
checkSituation(foundAcc)
add(accShadow)
add(accShadow)
add(accShadow)
add(account)
modify(accShadow)

*Figure 11: Reconciliation process.*

# 4 Verification

Section Solution Verification is divided into three subsections, Implementation, Used technologies and Evaluation. Subsection Implementation concentrates to describe concrete implementation of the mechanism which was made in the open-source identity management solution called midPoint. In this section are step by step described the individual methods needed for the consistency mechanism. In the next section, there are also stated used technologies. The last subsection, Evaluation, describes how the implemented mechanism was tested and results of the tests.

## *4.1 Implementation*

In this section is provided brief view on the concrete implementation of the mechanism in the midPoint solution. The main part of the implementation is placed in the provisioning component. After small refactoring of provisioning component introduced in detail in the Appendix C – Solution Design Diagrams we got the suitable place for the designed mechanism and it is the ShadowCache class.

This class first calls the ShadowConverter to propagate the changes to the external resources and if something went wrong one of the mentioned error is thrown (e.g CommunicationException, ObjectAlreadyExistException, ObjectNotFoundException, etc). All the thrown errors should be treated either only telling the user that the operation failed or trying to solve the problem.

The mechanism will rely on the UCF framework and will use its interface instead of ICF interface. UCF provides to us more friendly error handling where the ICF exceptions are wrapped to the few groups of exceptions. UCF also takes care for the result of operations (if they were successful or failed) and the hierarchy of called methods is placed in the OperationalResult which can be interesting for us by the reconciliation process.

The idea is that after the error is thrown the system chooses one of the mechanisms which will be suitable for the error. Therefore we decide that it will be implemented using the strategy design pattern. The concrete error handler will be constructed using the factory method according to the thrown error and then the mechanism for chosen error handler will be called.

Each error handler is implemented with respect to the error thrown. Error handlers for unprocessable errors are implemented in the simple manner and their responsibility stand for notifying the end user about the detected issue. Processable errors have specified more complex error handler. By the ObjectNotFoundException or ObjectAlreadyExistException the error handler tries to solve the inconsistencies immediately.

Resolution of inconsistencies implies success and the initial operation should continue. Otherwise, the error is thrown to the user to let him known about the inconsistencies. By the CommunicationException is the role of error handler to prepare objects in the repository for the reconciliation process. Reconciliation process will then use these objects to re-try the operations again. Proposed class diagram is shown in figure 12.

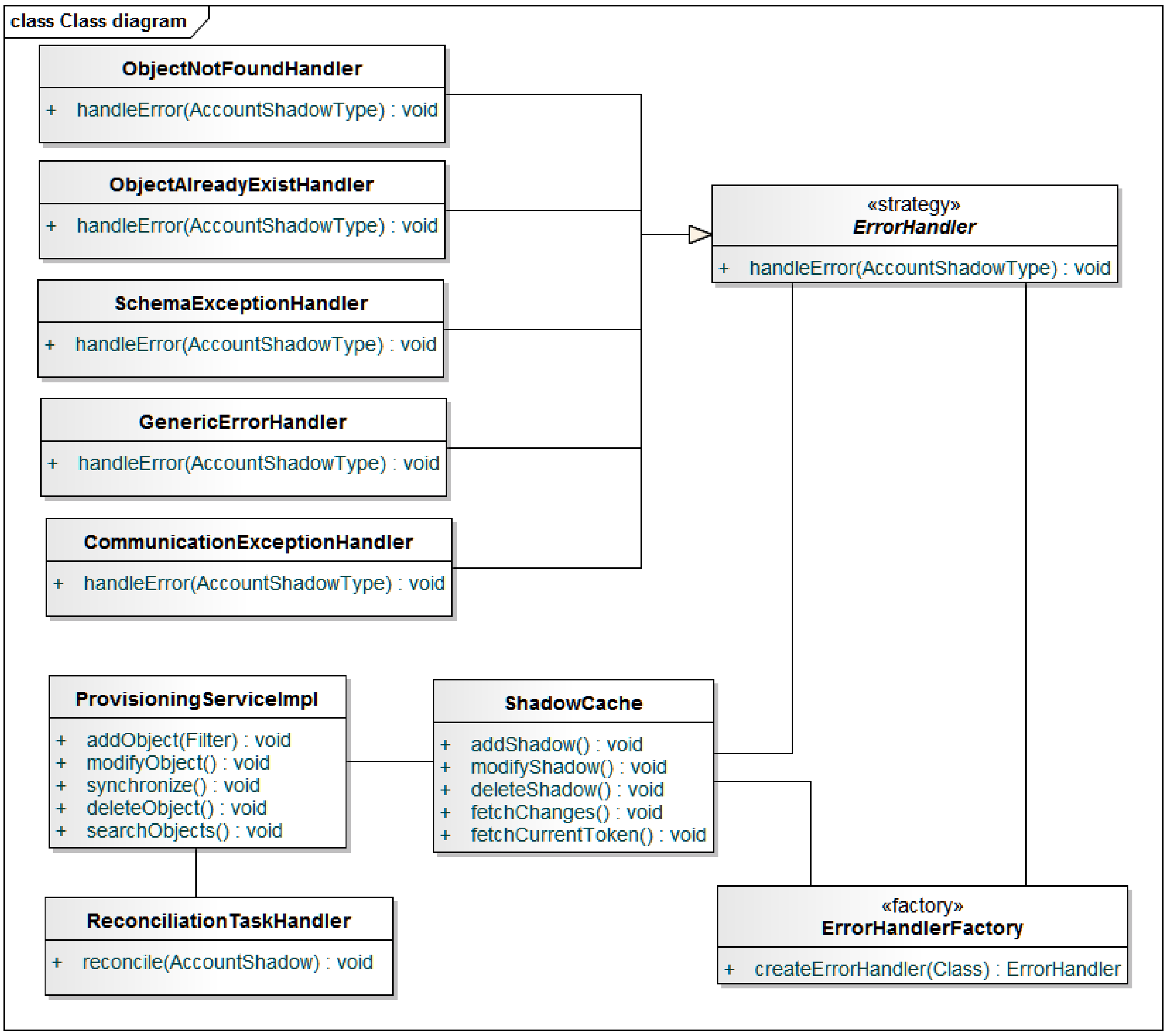

class Class diagram
ObjectNotFoundHandler
+ handleError(AccountShadowType) : void
ObjectAlreadyExistHandler
+ handleError(AccountShadowType) : void
SchemaExceptionHandler
+ handleError(AccountShadowType) : void
GenericErrorHandler
+ handleError(AccountShadowType) : void
CommunicationExceptionHandler
+ handleError(AccountShadowType) : void
«strategy»
ErrorHandler
+ handleError(AccountShadowType) : void
ProvisioningServiceImpl
+ addObject(Filter) : void
+ modifyObject() : void
+ synchronize() : void
+ deleteObject() : void
+ searchObjects() : void
ShadowCache
+ addShadow() : void
+ modifyShadow() : void
+ deleteShadow() : void
+ fetchChanges() : void
+ fetchCurrentToken() : void
ReconciliationTaskHandler
+ reconcile(AccountShadow) : void
«factory»
ErrorHandlerFactory
+ createErrorHandler(Class) : ErrorHandler

*Figure 12: Class diagram for proposed mechanism.*

Detailed view on each error handler and also other steps needed for the mechanism is detailed described in the next text.

### 4.1.1 Creation of Error handler

Creation of the concrete error handler is situated in the class ErrorHadlerFactory. The method createHandler() takes the Exception as the input. According to the Exception, the concrete instance of the ErrorHandler is constructed. Exceptions and related ErrorHandlers are stated in the table 5.

*Table 5: Exceptions and theirs error handlers.*

| Exception | ErrorHandler |
|---|---|
| CommunicationExcetpion | CommunicationErrorHandler |
| GenericErrorException | GenericErrorHandler |
| ObjectAlreadyExistException | ObjectAlreadyExistHandler |
| ObjectNotFoundException | ObjectNotFoundHandler |
| SchemaViolationException | SchemaViolationHandler |

However, there is no guarantee that the inputted Exception will be one of the above mentioned. In such situation ErrorHandler is not constructed but instead the SystemException is thrown. SystemExcpetion is instance of the RuntimeExcpetion and it is used, when we do not know the Error which occurrs.

### 4.1.2 Calling the Error handler

After the concrete instance of the ErrorHandler is created, it is needed to call it to perform the compensation to the error. Compensations are implemented in the method handleError() in each of the error handler separately. However, before the error handler is called, the correct values for the inputs must be set. This is done with the method setAdditionalParams() placed in the ShadowCache. Inputs for this method are:

- Shadow
  - by the add action, the whole account which should be created,
  - by the modify and delete action the actual shadow stored in the midPoint's repository.
- Failed operation type
  - describing the type of the operation that failed, ADD by add action, MODIFY by modify action, DELETE by delete action.
- Resource
  - identifier of the resource, where the account (should) exist(s).
- Operation result
  - hierarchically sorted summary of the steps in processing the concrete action with the account. In this summary there is information about which actions were called and what was the result of these actions. It also contains the reason what a why went wrong.
- Changes
  - description of changes which should be applied to the account. They are needed in the case of modify action failed.

Method setAdditionalParams() returns the shadow enriched with the additional information needed for compensating the error. Returned shadow is then passed as the input argument for the handleError() method.

### 4.1.3 Unprocessable errors

Generic errors and schema violation errors were classified as unprocessable errors to which compensations cannot be specified. For example, it can reflect that some of the mandatory attribute was not filled, or the shadow description does not satisfy the desired structure. By these errors theirs error handler is called, but the function of the handleError() method is only to throw exception and let the user known about the error. All the others steps are depended on the user decision.

### 4.1.4 Communication error handler

Responsibility of the compensation for the communication error is to store the processed account with all the additional information to the midPoint's repository. It provides preparation for the reconciliation process. There are small differences in attributes which need to be stored with the account depending on the concrete action executed (add, modify, delete). More implementation details and also additional attributes needed for individual action are introduced in the table 6.

*Table 6: Account's additional information to individual failed actions.*

| Action | Attributes stored in the midPoint's repository | Next steps |
|---|---|---|
| Add account | Whole account with all attributes, resource identifier, failed action, operational result | Link account to the user. Return success to the end user. |
| Modify account | Original account, changes made to the account, failed operation type, operational result | Return success to the end user. |
| Delete account | Original account, failed operation type | Unlink account form the user. Return success to the end user. |

As the table shows, the compensation to modify account is implemented to only store the changes to the account in the repository. In the other two operations, there are additional steps. If the addition of account failed, besides storing the account to the repository, there is also need to link this account to the concrete user. Opposite continuation, unlink account from user, is processed when the initial action was deletion.

Compensations to the communication errors are implemented in the manner to not tell end user, that the action may be unsuccessful. The end user gets the message that all his changed was done successfully. Even by the add action and delete action can end user see the changes immediately.

### 4.1.5 Object not found handler

Object not found handler contains definition for two different actions – modify account and delete account. First of all, if the method handleError() in the ObjectNotFoundHandler is called, there is decision which compensation should be called. Compensation for delete action calls the method to delete account shadow from the midPoint's repository. After deleting this account shadow from the midpoint's repository, the success is returned and then follows the action that unlinks non-existed account from the user.

Compensation for the modify action is little bit complicated. It results from the fact that it first must be decided, if the not found account should exist or not. For that reason, the synchronization mechanism is called to decide if the account will be re-created and the modifications will be applied, or the account will be deleted and unlinked from the user.

The account is re-created if the user has assignment[1] for such account. Newly created account is then linked to the user. Old account shadow is deleted from the repository and the link to the old account is removed from the user. Since the re-created account has a new identifier, applying the changes cannot be executed. We first need to find out the new identifier to be able to apply the changes.

When the account is re-created by the synchronization mechanism, the identifier is saved to the operation result as the return parameter. Then, in the compensation mechanism is this identifier retrieved from the operational result. Old identifier in the change description is replaced with the new identifier and the changes are applied calling the method modifyObject() from the ShadowCache.

No assignments imply deletion of the account. In addition, also the link to this account is removed from the user. In both cases the end user gets the message which closer describes the situation. If the account was re-created and the changes were applied, end user gets the *success* message, otherwise there is a message informing, that the changes were not applied because the account was not found.

### 4.1.6 Object already exist handler

Object already exist hander is a compensation for the action add account. This compensation is implemented in the two midPoint's components – provisioning and model.

1 Assignments are used to declare which role, group or account should user have. If user has assigned some account, this account MUST exist until the assignment exists. It means that if someone deletes such account without deleting assignment, the account should be re-created. If the assignment is removed, the account is deleted.

In the provisioning component, when the handleError() is called, the synchronization mechanism is invoked. Input for the synchronization mechanism is account from the resource[2]. It is obtained by the searchObject() method. Search filter for this method is set according to initial account's[3] identifier.

Synchronization mechanism performs the individual steps with the found account and results can be more. By running the synchronization mechanism it is possible that:

- the found account can be deleted from the end resource,
- the found account can be linked to this user,
- the found account can be linked to different user,
- new user is created to the found account.

Since it is not clear to the provisioning which action was actually performed, the provisioning cannot reasonably react and compensate the error. Therefore, after the synchronization mechanism run, the ObjectAlreadyExistException is re-thrown back to the model component. Implementation in the model component is responsible to invoke the initial action again, but with the new knowledge.

This results to the calling synchronizeUser() method to re-compute the changes which should be made. If the account was linked to the user by the synchronization mechanism run from the provisioning, there exists no more changes to be applied and the end user gets success message, that the account was created but with the new identifiers. If the account was deleted by the synchronization mechanism run in provisioning, the recomputed changes contain initial account which is tried to add again.

In the other two mentioned cases, the method sycnhronizeUser() re-computes the account's identifiers and the account is tried to add with these new identifiers. If new identifiers satisfy constraints, the account is added on the external resource and the end user gets the success message about the account which was added. Otherwise, the identifiers are re-computed again and again until constraints are not satisfied or the number of max iterations is not exceeded.

### 4.1.7 Reconciliation

Reconciliation consists of two steps. In the first step, the failed accounts are searched and then the failed action is invoked to be re-tried. Search is done in the midpoint's repository with the filter that satisfies only failed accounts. After the accounts are found, the failed action is invoked and runs again with the original parameters. The failed action which should be called is stored with the account in the midPoint's repository.

Reconciliation is either successful or failure. Success of the reconciliation process implies that the account shadow in the repository is cleaned. It means, that the additional

[2] This is an account, which already exists on the resource and we must decide what to do with this account.

3 Initial account is the account which should be created by the add action initiated from the end user.

information stored with the account are removed and there is stored only corresponding account shadow[4]. Such cleared account is then no more found by reconciliation process.

Failure of the reconciliation process implies increase for number of attempts. The account is processed by the reconciliation process only until the number of attempts exceeds the specified limit. After crossing the limit for the account, the midPoint's repository is cleaned of the account, e.g. if the failed account was formatted by the add action, the account is removed from the midPoint's repository and also unlinked from the user.

## 4.2 Used technologies

Mechanism was implemented using Java programming language and technology Spring and JAXB. The runtime environment for midpoint is application server Tomcat 6.0. Each of ErrorHandler-s is represented as a Spring bean and it is left to the spring container to carry out the life-cycle of the instances. JAXB technology is used for translating between XML and Java objects.

Automated test are implemented using a testing framework TestNG which provides functionality similar to JUnit test framework. The tests are provided against the XML repository called BaseX and SQL repository MySql. The external system used for testing is OpenDJ. Tests and mechanism was developed in the development kit eclipse.

## 4.3 Evaluations

Proposed mechanism was implemented and tested. Tests provided on the mechanism implementation in the first phase were done manually and in the second phase they were done automatically. Each action was simulated and the actual results were compared with the expected one. In the next sections the individual steps needed for manual and automatic tests are detailed described.

### 4.3.1 Manual testing

By the manual testing, the midPoint solution was deployed to the application server Tomcat 6.0. The prerequisites are also OpenDJ as an external system and BaseX as the local midPoint’s repostiory. The tests was provided by using Chrome browser.

The communication problem was tested in the way that we temporary stopped the OpenDJ and we try to add, modify and delete some accounts. Then we check the midPoint’s repository if it contains the additional information to these accounts. After the

4 In the midPoint's repository there are stored accounts that contains name, resource identifier, account type and the attributes which represents the resource account identifiers.

check and the success of the previous situations, the reconciliation process was tested whether it can repair previous inconsistencies. We first tested the reconciliation with the stopped OpenDJ.

The reconciliation is triggered after importing the reconciliation task. This task describes the reconciliation policies, e.g. how often the reconciliation is triggered, which resource should be reconciled etc. After the reconciliation run we check the repository again. There were the same accounts with the same additional information but the attempt number was increased. This is the expected result.

Then we started the OpenDJ and tested the reconciliation again. The results were as expected. After the OpenDJ was started, the failed operation was re-tried and they resulted with the success. We found this out after the checking the local midPoint's repository and also the state in the OpenDJ accounts. The local midPoint's repository contained only corresponding accounts without additional information.

The situation when object that is not found on the external resource is modified was also manually tested. In the first step in these tests we disabled synchronization of changes from resource to the midPoint. The tests were done so that we first got user and its accounts. Then we provided some modification of account's attributes and before the submit button was pressed, we deleted the account directly from the resource.

Then we expected two situations. If the user had the account created by assignment, we expected that the account will be re-created and the changes will be applied. In the other case (if the user had no assignment to the account) we expected that the account shadow will be deleted from the midPoint's repository and unlinked from the user. After checking the midPoint's repository we found out that both situations were performed as expected.

Addition of account that is already present on the external resource we also tested by the disabled synchronization of changes from external resource. We first intentionally harm the consistency of the data and manually add the account directly on the external resource. Then we created new user with the name satisfying the identifier of previously created account on the resource. After that we try to add new account to the user through the midPoint's web interface. We expected that the previously created account on the resource is linked to the user. And after checking the resource and also the midPoint's repository it was true.

Another example for situation that object already exists was manually tested. It describes the situation when we try to add the account that already exists on the external resource and is also linked to other user. In this case we expected, that the initial account's[5] identifiers are re-computed and the account is re-tried to add with these new identifiers. This situation also performs correctly.

Manual testing does not cover all the situations. It was rather performed to cover

[5] Account that we try to add.

the base examples and it should demonstrate the basic behavior of the mechanism. The more complicated scenarios were tested using automatic testing.

### 4.3.2 Automatic testing

The identified situations we also tested automatically. There was created project with the end-to-end tests. These tests simulate the end user’s actions and they are executed across the whole system. It means that instead of using web user interface and manually clicking desired actions, we use the model web service interface and write short functions to call desired actions. Also we do not use mock object, but real implementation.

Automatic tests use embedded OpenDJ instance as an end resource and there may be chosen the local midPoint’s repository. It can be SQL repository MySQL or XML repository BaseX. In the tests the synchronization of changes from the resource is disabled and there is also possibility to start and stop the embedded instance of OpenDJ as needed.

The other difference between manual and automatic tests is that we needed to prepare some example of objects (users, resource, accounts). These examples takes form of an XML and are used as inputs for calling the model web service. In the XML we describe how the object should look like, but also the actions that should be done. In the resource XML there are set reactions on the synchronization situations, for example. In the change description XML there is described the type of the change (modify, add, delete) and also the concrete attributes which should be changed (attribute’s name and also its value).

The concrete situation which was tested and also theirs results are detailed described in the  Appendix D – Test Results.

# 5 Discussion

In the previous sections we tried to design the algorithm which will be able to minimize the risk of inconsistencies and also it should be able to solve the problems and eventually brings the data to the consistent state. This mechanism was designed with respect to the facts, that the traditional transaction operations are not suitable, so it cannot be used any of the known mechanism for transactional systems.

The primary scientific contribution of this work is the possibility to perform the operations on the non-transactional systems with some of the benefits comparable to traditional transactions, bringing some consistency guarantees into such systems (with respect to the CAP theorem). We decided to use similar mechanism used by transactional systems that are able to recover from the errors. This mechanism attempts to solve the consistency issues by individual operation using the compensations and if it does not succeed after several attempts, it returns the data to the state before the operation was executed.

Proposed solution follows the model of the eventual consistency which means that the system does not guarantee that data will be consistent all the time. Instead, the temporary inconsistencies are allowed and the attention is made for the mechanism which solves the inconsistencies and eventually brings the data to the consistent state. The mechanism is based on the three base concepts:

- CAP theorem,
- relative change model,
- compensations for the unsuccessful operations.

CAP theorem shows that in the distributed system there cannot be guarantee for consistency availability and partition tolerance at once and it must be chosen only two of them. In our solution we choose availability and partition tolerance instead of strong consistency. This can be also called weak consistency model as we mentioned before.

We decided to weaken the consistency instead of availability because for such systems like identity management solutions are, it is required to guarantee high availability and so you can read and write to the system all the time. Every request to the system must have appropriate response even if failures occurs (e.g. one of the node is down). It does not matter if the operation was successful, but it must be terminated and the result returned to the user. Even, if a message sent from one node to another is lost, the system must continue to operate.

Another important concept for the proposed mechanism is a relative change model. Relative change model is used to describe changes made to the objects. Instead of sending the whole object even when only one of its attribute was changed, we send only the real change. The changes are computed with respect to the old object and the result of computation is the change description shown in the figure 13.

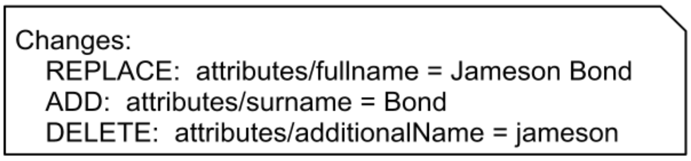

Changes:
REPLACE: attributes/fullname = Jameson Bond
ADD: attributes/surname = Bond
DELETE: attributes/additionalName = jameson

*Figure 13: Relative change model - changes description.*

In the figure 13, we can see the change of three attributes. The first parameter of the change is the change type. It describes if the attribute should be added, deleted of replaced with the new value. The second parameter describes the path to the attribute which should be changed and the last parameter is the value of the attribute. The changes are then patched and the new object is formed. It is shown in the figure 14.

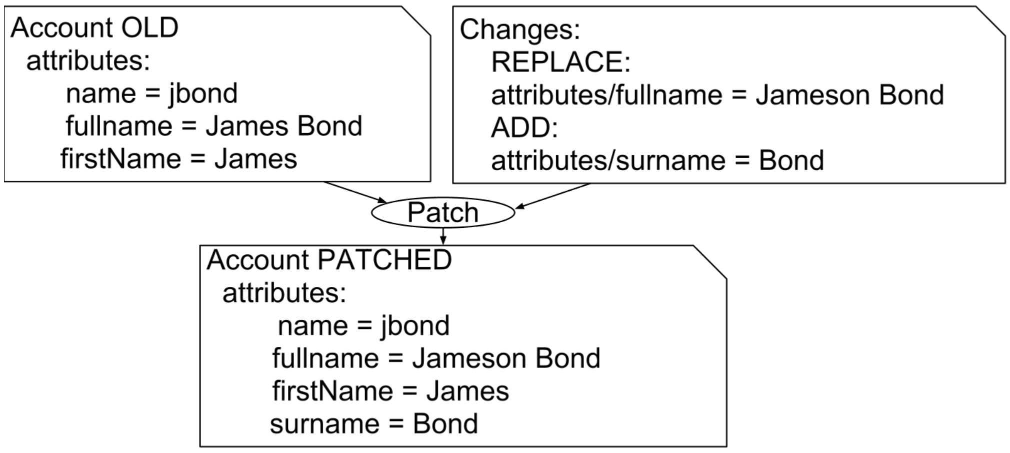


*Figure 14: Relative change model - patching changes.*

The advantage of using relative change model instead of absolute model is that the data do not need to be locked while they are used by some process. Also, if the object is modified with two different processes at almost the same time, the changes applied by the process which ends first may not be known to the other process and they will be replaced with new changed object. On the other side, if we are using relative change model we do not worry about such situation. Because not the whole object is replaced, but only the real changes.

Last concept that is important for the mechanism is compensations. Compensations are reactions for the errors. They are used for trying to eliminate the errors or to react to the errors and find appropriate way that will not harm the consistency constraints. Each executed operation can end successfully or unsuccessfully. If the operation ends with an error, first must be decided if the error is processable or not. Processable errors can have defined compensations, but unprocessable errors cannot. If the error is processable, then defined compensation mechanism is called to resolve the problem, otherwise the only think we can do is notify user about the error.

The mechanism is proposed to minimize the formation of the inconsistencies and if they ever happened, it should reasonably react and bring the system to the consistent state. The mechanism was designed with respect to the one of the identified groups of problems in the field of identity management – resource failures. It consists of two parts. The first part tries to handle the unexpected error that occurred and the second part, called reconciliation, is used to compare the state of the object stored in the repository and in the end resource.

It has to be known if the error that occurred is processable or not in the first part. If the error is processable, there are specified compensation mechanisms as the reaction to the error. Each error has its own compensation mechanism. Individual compensation mechanisms were described earlier in the section 3 Solution Design. If the error is not processable it means, that we do not know how to implement the compensation to the error. Such an error can be also considered as fatal, and then the user help is needed for its reparation. The example of the triggering compensation mechanism for failed operation is shown in the figure 15.

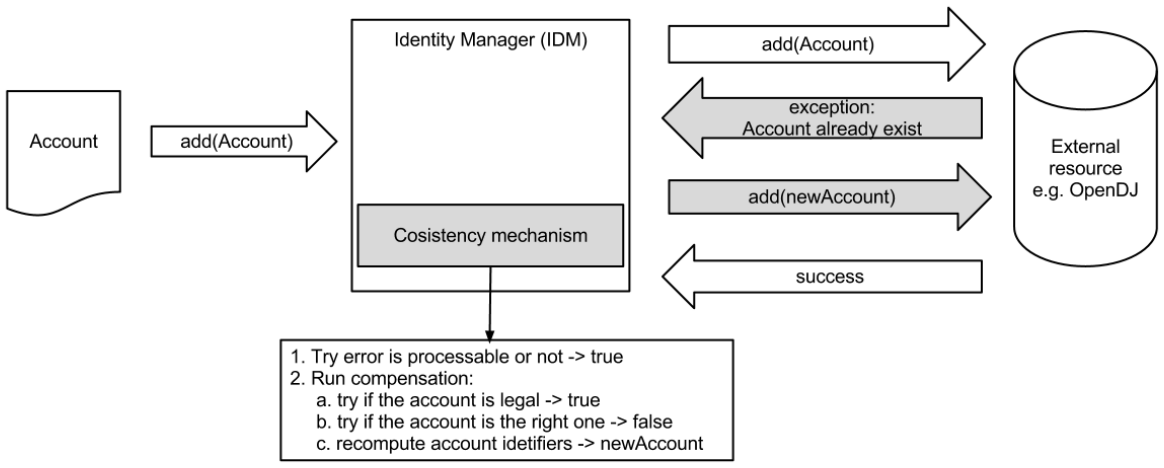


*Figure 15 The first part of the mechanism (compensation for the situation where the account already exist on the resource).*

Compensations can either eliminate error at the moment they originated, or it can postpone the error and try to eliminate it later. If the error is eliminated immediately, the result of the compensation should preserve the consistent state of the data. But, if the error cannot be eliminated immediately, it can harm the data consistency. Since we are not able to react immediately, we must store somewhere the error description.

Later, according to such stored error description it should be possible to find out what (operation), why (reason) and which data went wrong. Since it is not desired (especially in the systems like IDM are) to have inconsistencies in the data, there must be defined the way how can we return to this error and try to process it later. For that reason we proposed to use the reconciliation process.

Reconciliation process can be described as a process by which we can find differences among replicas on various resources. These differences are then merged to the one that is applied on other replicas. In our solution we propose to use the reconciliation process also to find previous errors and to trigger the operation which can eliminate the error. It should be executed in the regular interval but only a limited numbers of times. After the set number of attempts is exceeded, the data are returned to the state before that operation. The reconciliation process is shown in the figure 16.

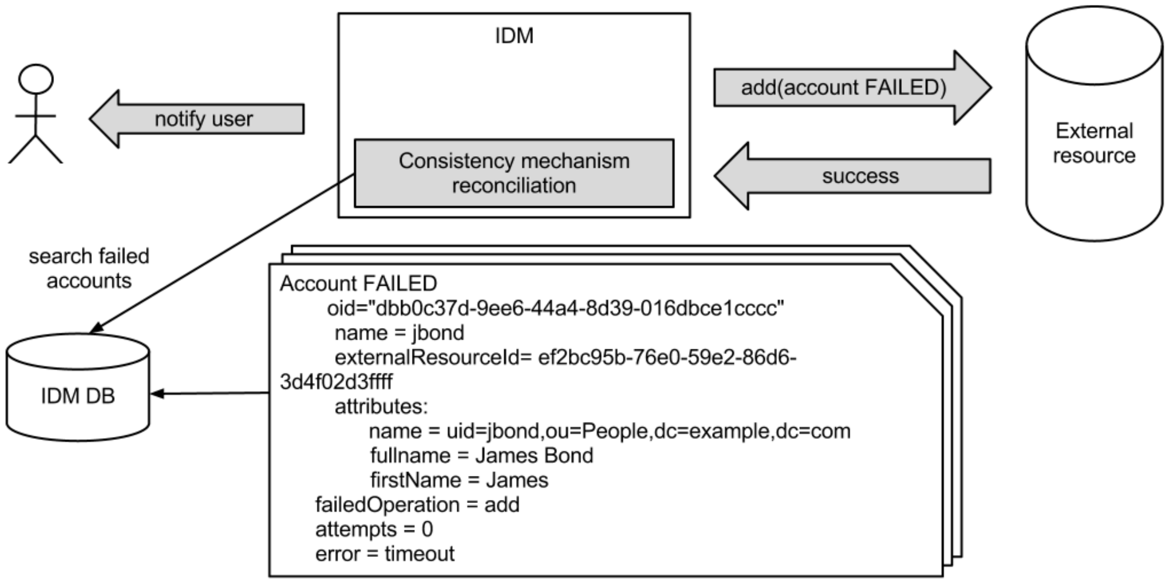


*Figure 16 Reconciliation process.*

It can’t be said that the proposed mechanism is perfect. It was designed with respect to only one of the defined groups of problems. So, there exists the possibility that it will need to be extended of some steps if considering also other defined groups of problems. If we consider inconsistencies which occur by the synchronization process, there should be a little bit different behavior. These operations are performed asynchronously and maybe it could be nice to tell the user if something went wrong and with the compensation it was postponed to be corrected later. Probably some notification mechanism will be needed.

Notification mechanism can be also good for other groups of problems. Imagine that the error is compensated and postponed to be handled later. If it is not successful after specific number of attempts, the data are returned to the state before the error was formed. The failed operation could be addition of account and the result is that the account was not added after specific number of attempts and the account was deleted from the local (identity manager) repository. End user has no possibility to know that the account was deleted. Notification should tell this to the end user and he could execute the action again.

We can also consider the inconsistencies which can occur while working with groups or dependencies. For example, LDAP has addition of account and assigning it to the group implemented as two different operations. There is not defined in the account that it belongs to the group, instead the group contains information about which accounts belong to it. However, addition of account and assigning it to the group consists of two different operations and it should be nice to provide it atomically or at least there must exist guarantee, that the result will be consistent.

# Conclusion

This thesis focused on the consistency in the identity management. At the beginning, the concept of identity management was introduced to explain the main principles and reasons for using identity management solution in the enterprise. Some of the main technologies used in the identity management solutions were introduced, such as single sign-on system, public key certificate, directory servers, LDAP or the provisioning engine. It was described that existing identity management solutions usually does not provide sufficient consistency guarantees. The aim of this thesis is to add a consistency guarantee to identity management solution. For that reason, known mechanisms used by variety of systems and the existing solutions of identity management were described.

Systems that were analyzed manage changes to maintain data in resources to be not contradictory by synchronization and reconciliation processes. But this is not enough. Another mechanism is needed to ensure that data after changes will be consistent. Because many of the resources managed by the identity management do not provide transactions, there cannot be used any of the known mechanism for ensuring consistency (like transactions). According to analyzed transaction mechanisms, one possibility is following the model of eventual consistency. It means that the data may be temporary inconsistent and we can only guarantee that eventually they will be consistent.

We decide to choose this way, so by designing the appropriate mechanism we were inclining to the eventual consistency. We designed the mechanism which was in detail described in the previous section (discussion). The mechanism is composed from two parts: error handling and reconciliation. It is also based on the three base concepts: compensations, CAP Theorem and relative change model. Pros and cons of the proposed mechanism are also concluded in the previous section (discussion).

Proposed mechanism was also implemented and tested. There were implemented automated tests and also the manual tests were executed. Automated test was implemented in the end-to-end manner and it simulated the actions by which the inconsistencies should occur. Automated tests were evaluated towards the expected results and they were successful. Although the automated tests run were successful and also by manual testing we get the expected results, it does not necessarily mean that the mechanism is perfect.

The mechanism was written in respect to one of the identified groups of problems – resource failures, and tests follow the issues in this group. If some operation of other identified group (synchronization problems, groups, role assignments, dependencies) fails, it is possible that the mechanism cannot reasonably react to the error and the inconsistencies occur. The next work on the mechanism should be focused on other identified situations by which inconsistencies may occur.

Proposed mechanism was also presented on the student's conference IIT.SRC. The implementation is a part of one existing open-source solution called midPoint and it will be included in the next release (2.0) of that project.

# Bibliography

[1] Abdallah, M., Guerraoui, R., Pucheral, P. 2002. *Dictatorial Transaction Processing: Atomic Commitment Without Veto Right*. Distrib. Parallel Databases 11, 3 (May 2002), 239-268.

[2] Bertino, E., Takahashi, K. 2010. *Identity Management: Concepts, Technologies and Systems.* ISBN 13: 978-1-60807-039-8. Publisher: artech house, publish date: December 2010.

[3] Brantner, B., et al. 2008. Building a database on S3. In Proceedings of the 2008 ACM SIGMOD international conference on Management of data (SIGMOD '08). ACM, New York, NY, USA, 251-264.

[4] Brewer, E. 2010. A certain freedom: thoughts on the CAP theorem. In Proceeding of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing (PODC '10). ACM, New York, NY, USA, 335-335.

[5] Ferraiolo, D.F., Kuhn, D.R. 1992. Role Based Access Control. 15th National Computer Security Conference. pp. 554-563.

[6] Garcia-Molina, H., Salem, K. 1987. Sagas. In Proceedings of the 1987 ACM SIGMOD international conference on Management of data (SIGMOD '87), Umeshwar Dayal (Ed.). ACM, New York, NY, USA, 249-259.

[7] Gilbert, S., Lynch, N. 2002. *Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services*. SIGACT News 33, 2 (June 2002), 51-59.

[8] Gray, J. 1988. *The transaction concept: virtues and limitations*. In Readings in database systems. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA 140-150.

[9] Greenfield, P. et al. 2003. *Compensation is Not Enough*. In Proceedings of the 7th International Conference on Enterprise Distributed Object Computing (EDOC '03). IEEE Computer Society, Washington, DC, USA, 232-240.

[10] midPoint community. Project wiki page: http://wiki.evolveum.com/display/midPoint/

[11] Neuman, T., et al. 2005. *The Kerberos Network Authentication Service (V5).* RFC 4124, Internet Engineering Task Force (2005).

[12] OpenIAM community. Project wiki page: Developer's documentation. http://wiki.openiam.org/dashboard.action

[13] OpenIAM community. Issue tracking tool Jira. http://jira.openiam.org/secure/Dashboard.jspa

[14] OpenICF community. Project wiki page: http://openicf.forgerock.org

[15] OpenIDM community. Project wiki page: Architecture guide. https://wikis.forgerock.org/confluence/display/openidm/Home

[16] OpenPTK community. Project wiki page: Overview. http://wikis.sun.com/display/openptk/Docs+2.0+Overview

[17] Pashalidis, A., Mitchell, C. 2003. *A taxonomy of single sign-on systems.* Proceedings, volume 2727 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, (July) 2003, pp. 249-264.

[18] Pfitzmann, A., Hansen, M. 2010. *A Terminology for Talking About Privacy by Data Minimization: Anonymity, Unlinkability, Undetectability, Unobservability, Pseudonymity, and Identity Management.* (August) 2010. 1 – 98.

[19] Pritchett, D. 2008. *BASE: An Acid Alternative*. Queue 6, 3 (May 2008), 48-55.

[20] Sandhu, R., et al. 1996. *Role-Based Access Control Models.* Computer, vol. 29, no. 2, pp. 38-47, Feb. 1996.

[21] Semančík, R. 2006. *Choosing the Best Identity Management Technology for Your Business.* Proceedings of InfoSecOn 2006 Conference, Cavtat, Croatia, 2006. 1- 10.

[22] Semančík, R. 2005. *How to Deploy Digital Identity Technology.* Proceedings of Network Forum 2005 Conference, Banská Bystrica, Slovakia, pp. 48-56, 2005

[23] Slone, S. and the open group identity management work area. 2004. Identity Management. 1-109.

[24] Wahl, M., Howes, T., Kille, S. 1997. *Lightweight Directory Access Protocol* (v3). RFC 2251, Internet Engineering Task Force (1997).

[25] Wang, T., et al. 2008. *A survey on the history of transaction management: from flat to grid transactions*. Distrib. Parallel Databases 23, 3 (June 2008), 235-270.

[26] Wolfson, O. 1987. *The overhead of locking (and commit) protocols in distributed databases*.ACM Trans. Database Syst. 12, 3 (September 1987), 453-471.

# Appendix A - Requirements Analysis and Specification

## A.1 Situations

In this section detailed view on the situations which can lead to the inconsistencies is provided. These situations were introduced in the section 2.1 Identified situations and this section should give the better understanding of the problems. The individual operations of identified groups of problems are also shown in the figure 17 and in the next text they are described in detail.

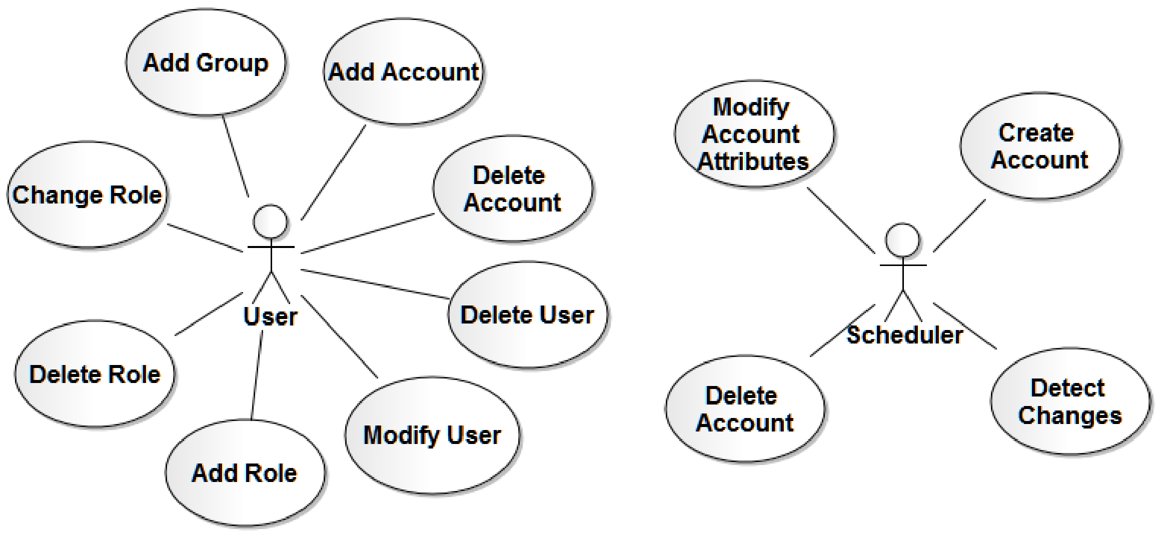


*Figure 17: Use cases by which inconsistencies should occur.*

### A.1.1 Resource failures

User's data are stored in the local repository and in the repository of external resource. If new account for the user is created, it should be stored to the local repository and also to the repository of the external resource. A situation may occur when changes are saved only in the external resource, but not in the local repository, e.g. account was created on the external resource, but the connector generated some exception (e.g. timeout,...). Identity manager doesn't know that the account exists, so the end user gets the error message stating that creation of the account failed. However, account exists in the resource but was not propagated to the local repository of identity manager. This is inconsistent state and it is needed to be eventually resolved.

Other example should be deletion of the existing account. Account is deleted from the external resource but connector generated some unexpected error. Because of this error, the account is not deleted from the local identity manager repository and the user still has reference to the non-existing account. Identity manager does not know that the account was actually deleted from the external resource but it was not deleted from the local repository. The user gets again the error message stating that the account was not deleted. This is also inconsistent state and it is needed to be resolved.

Also, when user is deleted, his accounts should be deleted from resources and from local repository, too. A situation may occur when user was deleted but some of his accounts were left behind. In addition, some resources may support additional operations when creating, deleting or modifying user, e.g. creation of new home directory and so on. The error may occur while executing this additional operation and it may cause the illusion that the main operation has failed. Connector reports the error, account should be changed, but the identity manager doesn't know it. The end user again gets the error message that data are in inconsistent state which again needs to be resolved.

### A.1.2 Dependencies and Groups

Some inconsistencies may occur when creating account dependent on other accounts. Such an example might be creating an account in some application depending on the account in the operation system. If the creation of the account in the operation system failed, it has no sense to create account in the application.

Example of situation when inconsistency occurs may be also adding account to the group. For example creation of new account in the LDAP databases and adding it to the group are two different operations. It is needed to solve how the identity manager should behave if the group does not exist e.g. if an exception should be thrown indicating that the group in which the account is trying to add doesn't exist or create this group and add account to it.

### A.1.3 Role Assignment

Role in the identity management doesn't specify only user's rights but they also specify which accounts should be created for the user. Some role might have specified that account on the four different external resources should be created. While creating these accounts, an unexpected error may occur. This causes that instead of four different accounts on the four different resources, only two are created. It means that the role is not complete and it is inconsistent state. Role shouldn't exist in the partial state and this is needed to be solved.

Employees usually change their work positions. For example the employee with the sales role was promoted to the manager role. Not only was his specialization changed, but also his rights and access to the connected resources. It means that role in the identity manager should also be changed. Inconsistent state can occur by changing role when account belonging to the old role is not deleted properly, or not all accounts for the new role are created.

Similarly when deleting user's role, some accounts might not be deleted which results in the inconsistencies. The use cases related to the consistency issues while propagating the changes from the identity management to the target system are shown in figure 8.

### A.1.4 Synchronization

Synchronization is process by which the changes done on the external resource are propagated to other connected resources and also to local identity manager repository. Since delivery of synchronization notifications may fail, some of the changes may not be detected. This may result in the permanently inconsistent state.

Inconsistent state might occur also when creating account on the external resource. The change of adding new account was detected and propagated to the identity manager. Identity manager tries to propagate changes to local repository, but it failed (some error occurred that made it impossible to create user in the identity manager repository). Account on the external resource exists, it was created also in the local repository, but no corresponding user was created.

Other example when synchronization results to the inconsistent state might be deleting of the account on the external resource. As in the previous example, changes are propagated to the identity manager, but by processing the changes some exception occurred, the user is not deleted and the reference to the account still exist. User is in the inconsistent state and it is needed to be solved.

Change of account's attributes on the external resource might also result in the inconsistencies. If some attribute was changed in the external resource and this attribute is describing user, it should be propagated and changed also in the local repository. If some error occurred while changing such an attribute, it results to the inconsistent state. The use cases related to the consistency issues which can be caused by synchronization process are shown in figure 9.

## *A.2 Use cases*

In this section there are in detail described individual use cases identified in the section 2.3 Use case model. The figure 18 shows which use cases are considered by designing the mechanism.

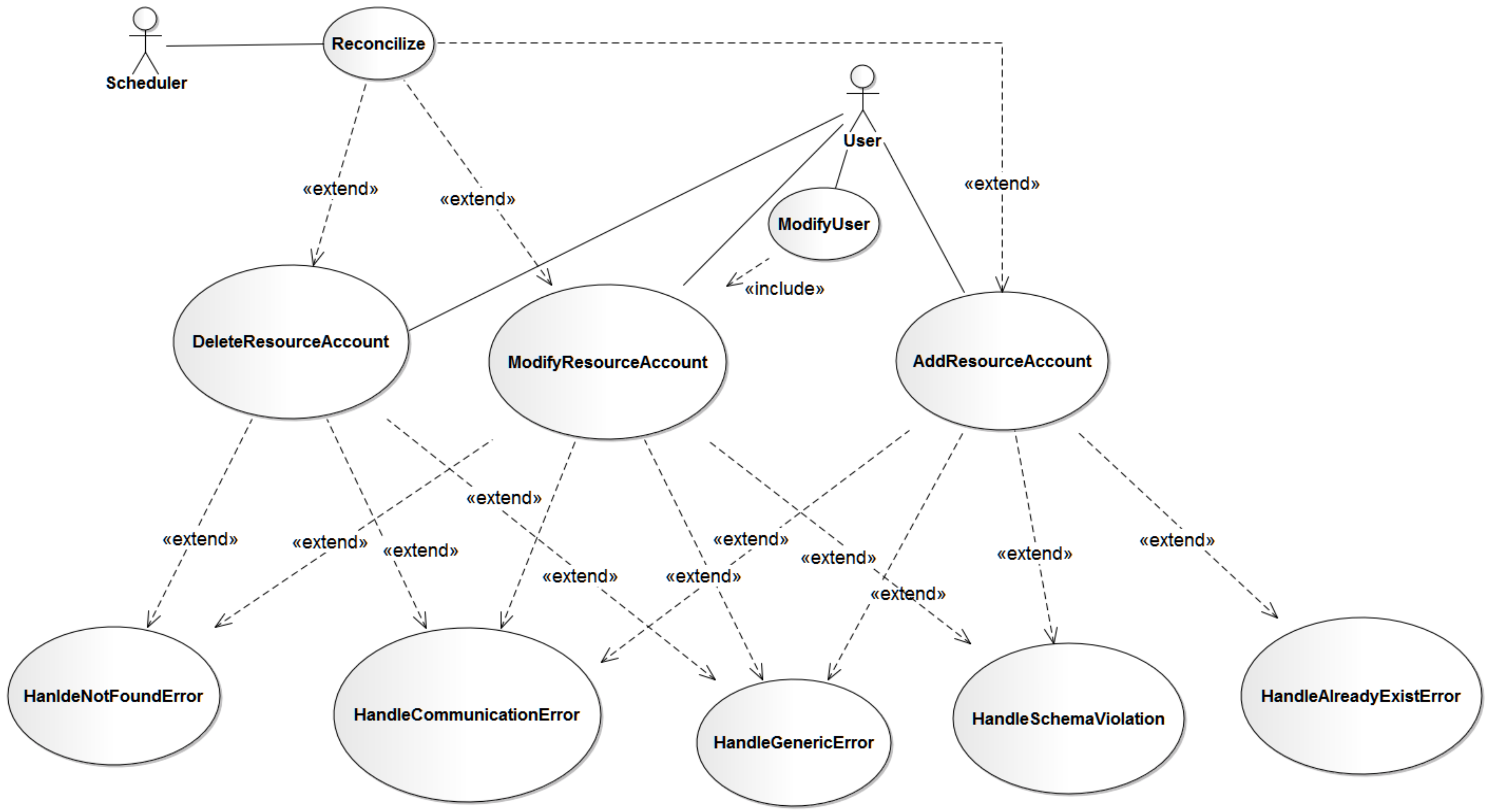


*Figure 18: Use case model.*

### A.2.1 Add Resource Account

Add resource account use case describes the action when the end user tries to add account through the identity manager user interface. The creation of account might be automated, e.g. by using user's attributes or it is created according to the user inputs. This created account is provisioned to the target system. There may happen different situations why the account was not added and another use cases is therefore called:

- If some problem with the end user's specified attributes occurs, the use case *HandleSchemaViolation* is started.
- If some problem with the communication with the external resource occur, e.g. the resource is down, the resource throws timeout exception etc, the use case *HandleCommunicationError* is started.
- If the situation that the account actually already exists on the external resource occurs, the use case *HandleAlreadyExist* is started.
- If a different error occur, e.g. by SQL database there was bad SQL syntax constructed, the use case *HandleGenericError* is started.

This extending use cases try to find out the error and call appropriate operation to solve the problem or to prevent from inconsistencies. In some cases, there is no possibility for the system to repair the state of the created account to be successfully added and the only possibility is to tell the end user to correct his inputs. On the other hand, some of the errors may be repaired either immediately or later by reconciliation process. If the extended use case ends, the end user gets the message what happened in the system. According to the message, the end user chooses the further action. The use case ends.

### A.2.2 Modify User, Modify Resource Account

Modify user use case includes the modify resource account use case. It is influenced by the fact that if the user was modified (e.g. modification of his name, or last name etc.) this change must be mapped to the account's attributes change. Then by modifying account attributes either according to user's changes or directly by account's attributes changes defined by end user, this changes are propagated to the external resources. The external resource might notice some issue why the accounts cannot be modified and the extending use cases are called:

- If some problem with the modifying attributes occurs, the use case *HandleSchemaViolation* is started.
- If some problem with the communication with the external resource occur, e.g. the resource is down, the resource throws timeout exception etc, the use case *HandleCommunicationError* is started.

- If the situation that the modifying account was not found on the external resource occurs, the use case *HandleNotFound* is started.
- If a different error occurs, e.g. by SQL database there was bad SQL syntax constructed, the use case *HandleGenericError* is started.

This extending use cases try to find out the error and call appropriate operation to solve the problem or to prevent from inconsistencies. In some cases, there is no possibility for the system to repair the state of the modified account to successfully modify its attributes in the external resource and the only possibility is to tell end user to correct his inputs. On the other hand, some of the errors may be repaired either immediately or later by reconciliation process. If the extended use case ends, the end user gets the message what happened in the system. According to the message, the end user chooses the further action. The use case ends.

### A.2.3 Delete Resource Account

Delete resource account use case is triggered either by deleting user (because if the user is deleted from the system all his accounts must be deleted from local repository and external resource, too) or directly by deleting concrete user's account. However was delete resource account use case triggered, it means that this deletion requirement is propagated to the target system where must be corresponding account deleted, too and this can leads to trigger the following use cases:

- If some problem with the communication with the external resource occur, e.g. the resource is down, the resource throws timeout exception etc, the use case *HandleCommunicationError* is started.
- If the situation that the modifying account was not found on the external resource occurs, the use case *HandleNotFound* is started.
- If some another error occur, e.g. by SQL database there was bad SQL syntax constructed, the use case *HandleGenericError* is started.

This extending use cases try to find out the error and call appropriate operation to solve the problem or to prevent from inconsistencies. In some cases, there is no possibility for the system to recover from the error and the account can't be deleted. On the other hand, some of the errors may be handled later by reconciliation process. If the extended use case ends, the end user gets the message what happened in the system. According to the message, the end user chooses the further action. The use case ends.

### A.2.4 Reconciliation

The use case reconciliation is used to describe the mechanism which will be responsible to detect the inconsistencies state of the system and try to solve them. According to the comparison of the local repository and the external resource it finds the problems and tries to solve them. The reconciliation will be triggered by the system and it will be run in the regular intervals. The results will be the situation which should be triggered by the system to repair the inconsistencies.

# Appendix B - MidPoint product description

### B.1.1 Basic Principles and Supported Features

The development of midPoint is governed by set of principles that reflect to the business needs in the identity management field as the [10] states. These principles are:

- Open system – midPoint is an open-source project, it is developed under CDDL license, the source code is public, only open protocols and platforms are used and it can be used for anyone for any purpose at no charge.
- Efficient common case – the idea is to focus on the specifics of the deployment rather than doing the same thing again and again for each deployment, e.g. data transformation can be implemented as one-line expression, setting up the new external resource is represented in only few clicks, there are auto-generated forms for the target systems which brings dynamical user interface.
- Extensible as needed – advanced scenarios can be supported by extending the code. Naturally, Java is supported, but there are plans also for Groovy, BPEL and other practical languages that the engineers should need.
- Data unification – as each system has its own model for identity data, security etc. the midPoint tries to provide a common data model for enterprise identity and so make the integration easier.

At the time when this part of the thesis was written, the latest midPoint version was release 1.9 therefore that version is considered in this section. According to [10] this release provide following features:

- basic provisioning from midPoint administrator's point of view to the external resources (create, read, update, delete),
- integration of Identity Connector Framework, which is used for connection to the external resources,
- identity repository based on BaseX XML database,
- live synchronization to propagate changes from external resource to the midPoint,
- support for the XPath version 2 expressions,
- enhanced logging and error reporting by introducing OperationResults,
- basic task manager component,
- basic Role Base Access Control (RBAC) and assignment,
- lightweight structure,
- support for Apache Tomcat web container,
- import from file and resource.

### B.1.2 Consistency and Release 1.9

Release 1.9 doesn't provide any consistency guarantees, there are no consistency features. However, midPoint wiki page contains some ideas about consistency. The main idea is to follow the weak consistency model [10]. This means that there doesn't exist guarantee that the data will be consistent all the time, but the proposal for the solution to the mechanism which would be responsible for  preventing system from inconsistencies or that should resolve consistency issues hasn't been made yet.

Two mechanisms for improving data consistency were proposed on the midPoint wiki page [10], relative change model and optimistic locking. The optimistic locking is only mentioned and no ideas about it are introduced. Some attention is made for the relative change model and its brief overview about what it means. Relative change model introduces the idea that only changes of properties are described and properties which were not changed are not specified. This model seems to provide easy merging and in vast majority of cases provide acceptable consistency without the need for locking.

All objects in the midPoint are assumed to be an XML objects. Based on the relative change model, the objects can be added or deleted and the objects properties values can be added, replaced or removed. Replacing is possible only for the whole properties values, not for single character in value. In the case, when only the single character is changed, e.g. property *foo* to *foa*, the *foo* property must be removed and new value *foa* must be added. In the midPoint there are three operations that implement changes [10]:

- Add object – this operation will add new object with the identifier and the property values.
- Modify object – this operation will modify specific properties of existing objects. Actually, there are three sub-operations:
    - Add property values – add new property values to the existing property values and has no impact on the existing properties.
    - Remove property values – remove specified values and has no impact on other properties.
    - Replace property values – remove existing values and replace them with the new set of values.
- Remove object – this operation remove specified object.

It is desired to have all these operations implemented as atomic what means that operation is either success and the changes are durable or failure and the changes are not applied. By some of the storage mechanisms it is possible to implement these operations atomically, but some of them may not be able to provide atomicity or might require short transactions. Atomicity and consistency of object by provisioning them to the target system is missing and it is open issue as the midPoint wiki [10] sates.

The midPoint wiki page [10] also contains comparison of relative change model and absolute change model. The disadvantages of absolute change model are explained on the importance of operation order. If the absolute change model would be used, the objects or its properties must be locked to provide reasonable degree of the consistency. By the long running processes that are common for the identity management systems (e.g. approvals) this lock will cause that no other changes can be applied until the previous process ended. This approach brings the bottleneck to the systems especially for the frequently used properties such as roles.

Relative change model also doesn't provide the wholly guarantee that all operations will be successful and no failures which can result to the inconsistencies occur. There are situation when it may fails and then the appropriate mechanism to handle conflicts and errors must be used.

## B.2 Provisioning Subsystem and its Current State

As mentioned before ( 1.3.4 MidPoint), midPoint is divided into components which can be built together as Lego bars. For the purpose of this thesis the provisioning component of midPoint is the primary point of the interest. This is a part responsible for talking to the external resources. Its responsibility is to manage identity related objects like accounts, groups, roles, etc. and propagates them to the external resources as well as to the local midPoint repository. Provisioning component is also responsible to detect changes on external resources and propagate them back to the midPoint.

All the logic about handling the accounts, groups, roles, entitlements or other identity related objects is situated in this component. Therefore it seems to be the suitable place where to have mechanism that should minimize the formation of inconsistencies. Obviously, such mechanism cannot be responsible only for minimizing risk of inconsistencies but it must be also able to reasonable react to such situations and produce the solution to consistency issues.

The handling of shadows like theirs creating or modifying and transferring among midPoint and external resources is based on the outbound and inbound expressions. These expressions are configured in the resource description and they define the policies how the account attributes should be transferred between systems as shown in figure 19. Inbound expression describes account's changes which should be transformed to the user change and on the other hand, outbound expression describes the user's changes which should be transformed to the account change.

Administrator can also set up the policies for synchronization and so he can influence what should happen if the new change is detected. These policies are situated also in the resource description. The connection to the external resource is made using Identity Connector Framework (ICF) which was originally developed under Sun Microsystems, and now it is supported by Forgerock under the OpenICF name [14].

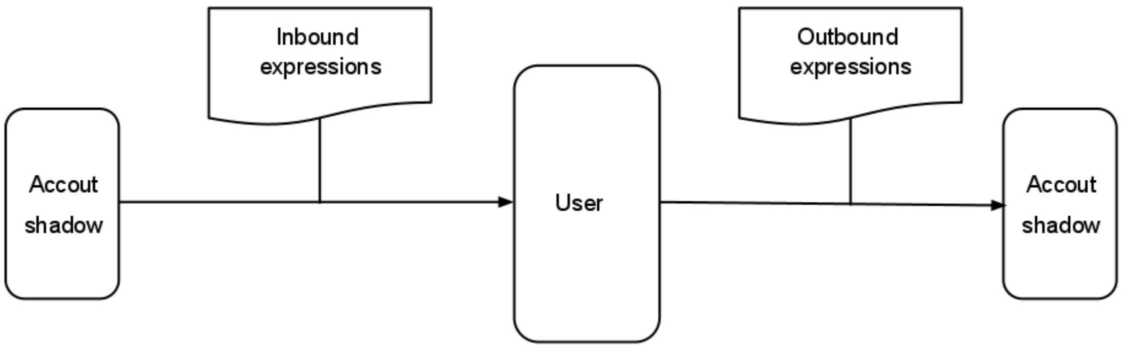


*Figure 19: Inbound and Outbound expressions.*

Because of some shortcomings of Identity Connector Framework (e.g. error handling)[6], new layer between ICF and midPoint with the name Unified Connector Framework (UCF) was developed which is supposed to provide more friendly error reporting. For the purposes of this thesis it is very important to have good error reporting, because it is needed to know where the error occurred and what the reason was.

Also new mechanism for collecting operation results was already presented in midPoint. This mechanism allows for remembering the operation, its parameters and also the status of the operations if it was successful or failed. Failure status also provides information about the error which occurs and the reason for the error. It was expected that the work presented in this thesis can take advantage of this existing mechanism.

Only provisioning of the accounts is implemented, implementation of groups and other objects or entitlements needed for identity management is missing. Also there is no implementation of the roles and its assignment at this time. No guarantee on consistency state of objects is made and it is also missing in the midPoint design. Therefore, the thesis will next concentrate to design appropriate mechanisms which will be able to solve the problem. With respect to the release 1.9 the provisioning component consists of the classes shown in the figure 20.

Each of the classes shown in figure 20 has its own responsibility. The ProvisioningService class should be considered as the facade provided to the other midPoint subsystems and it is responsible to route interface calls to the appropriate place. Except few cases where resource objects are needed, e.g. for diagnostic purposes, the ProvisionigService is supposed to deal with the XML objects. Other responsibilities are for resource configuration, changes detection on external resources and providing live synchronization and discovery [10].

6 http://wiki.evolveum.com/display/midPoint/ICF+Issues

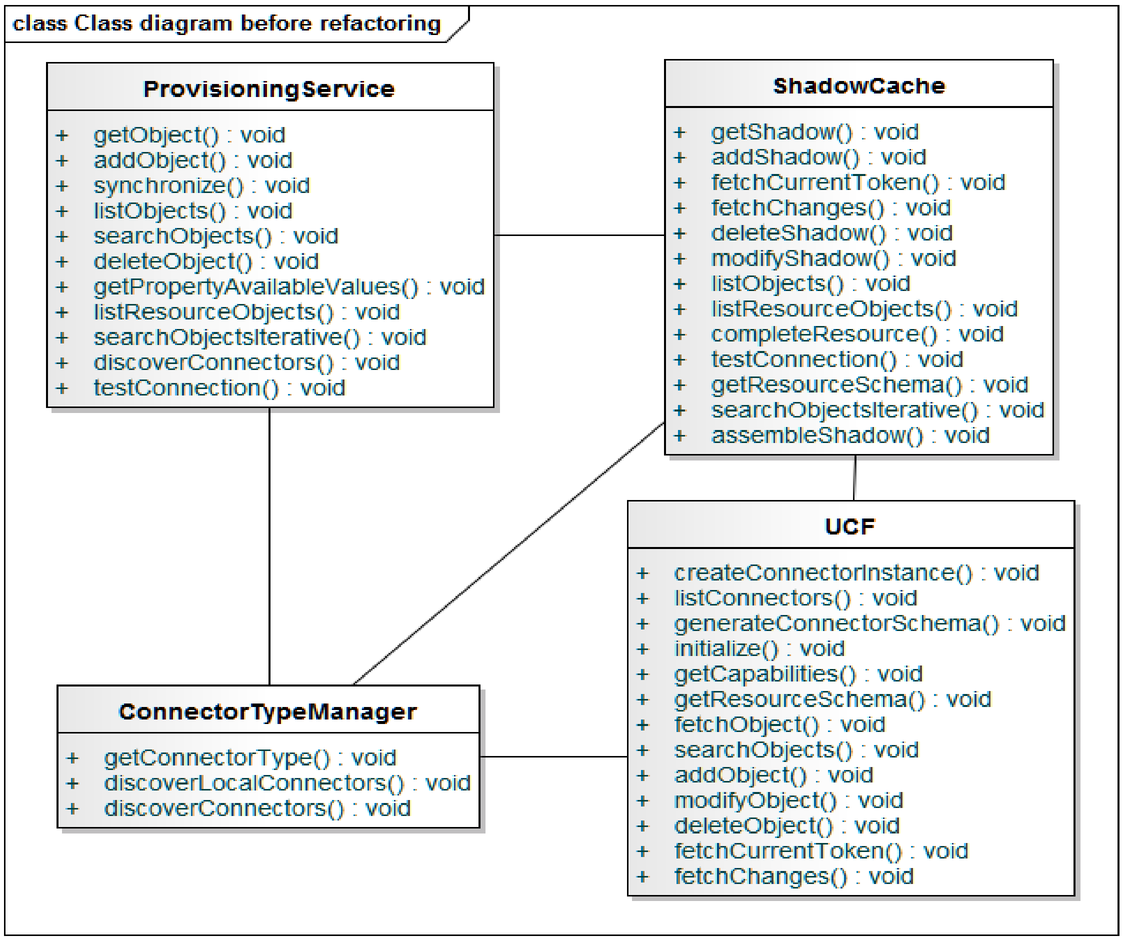

class Class diagram before refactoring
ProvisioningService
+ getObject() : void
+ addObject() : void
+ synchronize() : void
+ listObjects() : void
+ searchObjects() : void
+ deleteObject() : void
+ getPropertyAvailableValues() : void
+ listResourceObjects() : void
+ searchObjectsIterative() : void
+ discoverConnectors() : void
+ testConnection() : void
ShadowCache
+ getShadow() : void
+ addShadow() : void
+ fetchCurrentToken() : void
+ fetchChanges() : void
+ deleteShadow() : void
+ modifyShadow() : void
+ listObjects() : void
+ listResourceObjects() : void
+ completeResource() : void
+ testConnection() : void
+ getResourceSchema() : void
+ searchObjectsIterative() : void
+ assembleShadow() : void
UCF
+ createConnectorInstance() : void
+ listConnectors() : void
+ generateConnectorSchema() : void
+ initialize() : void
+ getCapabilities() : void
+ getResourceSchema() : void
+ fetchObject() : void
+ searchObjects() : void
+ addObject() : void
+ modifyObject() : void
+ deleteObject() : void
+ fetchCurrentToken() : void
+ fetchChanges() : void
ConnectorTypeManager
+ getConnectorType() : void
+ discoverLocalConnectors() : void
+ discoverConnectors() : void

*Figure 20: Provisioning classes with respect to the release 1.9.*

ShadowCache class provides the connection to the external resources using UCF. It deals with the resource objects as well as the XML objects. Its responsibility is conversion between native resource objects and midpoint's XML objects. Objects that shadow cache handles are partially saved in the local repository and partially on the external resource. These objects must be merged to the single, consistent view with aligning the representation of shadow objects in local repository and resource objects on the external resources. And this is what the shadow cache may do [10].

UCF stand for the Unified Connector Framework and it was designed to expand and unify capabilities of existing connecting framework. This should be a solution to some ICF issues[7]. It is still work in progress, so it should be not publicly visible. UCF deals with the resource objects and it knows nothing about the shadow objects. The ConnectorTypeManager is responsible for managing connector object in repository, like creating new connector object when a new local connector is discovered, takes care of remove connector discovery etc. [10].

7 http://wiki.evolveum.com/display/midPoint/ICF+Issues

## B.3 Data Model and Representation

The objects in midPoint take the XML form. In fact, they should be represented also using other custom format or even JSON because the midPoint's data model tries to be independent on the data representation. Therefore the objects are most frequently represented as Java objects in the computer memory. Objects can be persistently stored and identified with generated object identifier (OID) which is supposed to provide the uniqueness. Also the name of the object is used to identify the object with the more human-friendly description. The objects in midPoint should be of the various types expected in the provisioning system, e.g. user, account, resource etc. [10].

The main structure in the midPoint is the Object that can be extended to the concrete type, e.g. account, resource, etc. All objects consist of properties which are supposed to be understandable unit for the midPoint. Property can be single-valued or multi-valued. It is for example the name of the account, the account attributes or the resource host name etc. Mostly, properties are represented as a string data, but they should take also the form of integer, enumeration or even the complex type [10].

For the thesis purposes, the account shadow objects are most interesting. With the respect to the midPoint release 1.9 account shadow object is extension of resource shadow object. It has defined OID, name, attributes, credentials, activation properties and reference to the resource where it should exist. Account attributes which are saved in the midPoint repository are only those used to identify this object on the external resource. According to this identifying attributes it is possible to find the whole account on the external resource and give the account to the end user. The link between the account and the concrete user is placed in the user object.

Resource object is used to define the external resource where the account should exist. This resource object provide the definition about the schema, where it can be find which attributes the account is supposed to have. Resource object also contain the configuration part, where the suitable connector is set and also synchronization part is placed here. The synchronization part tells which different actions should occur and define the reactions to these actions. Besides this, it can be defined also the outbound and inbound expressions in the resource object. According to them, the account should be automatically generated from the user's properties. The part of the data model related to the provisioning and to the thesis purposes is shown in figure 21.

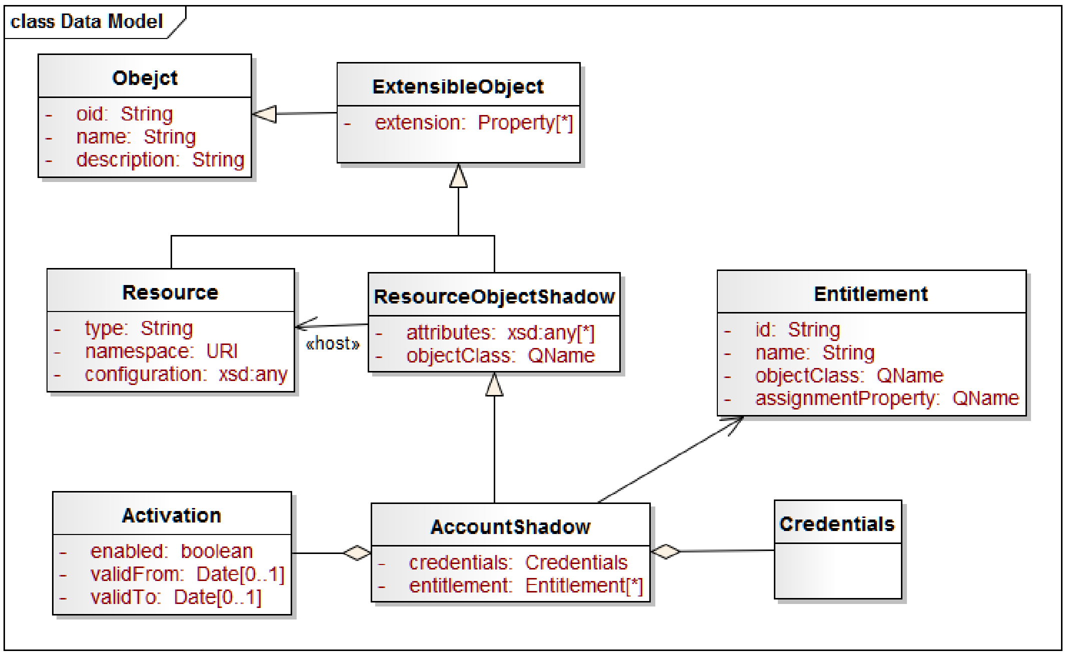
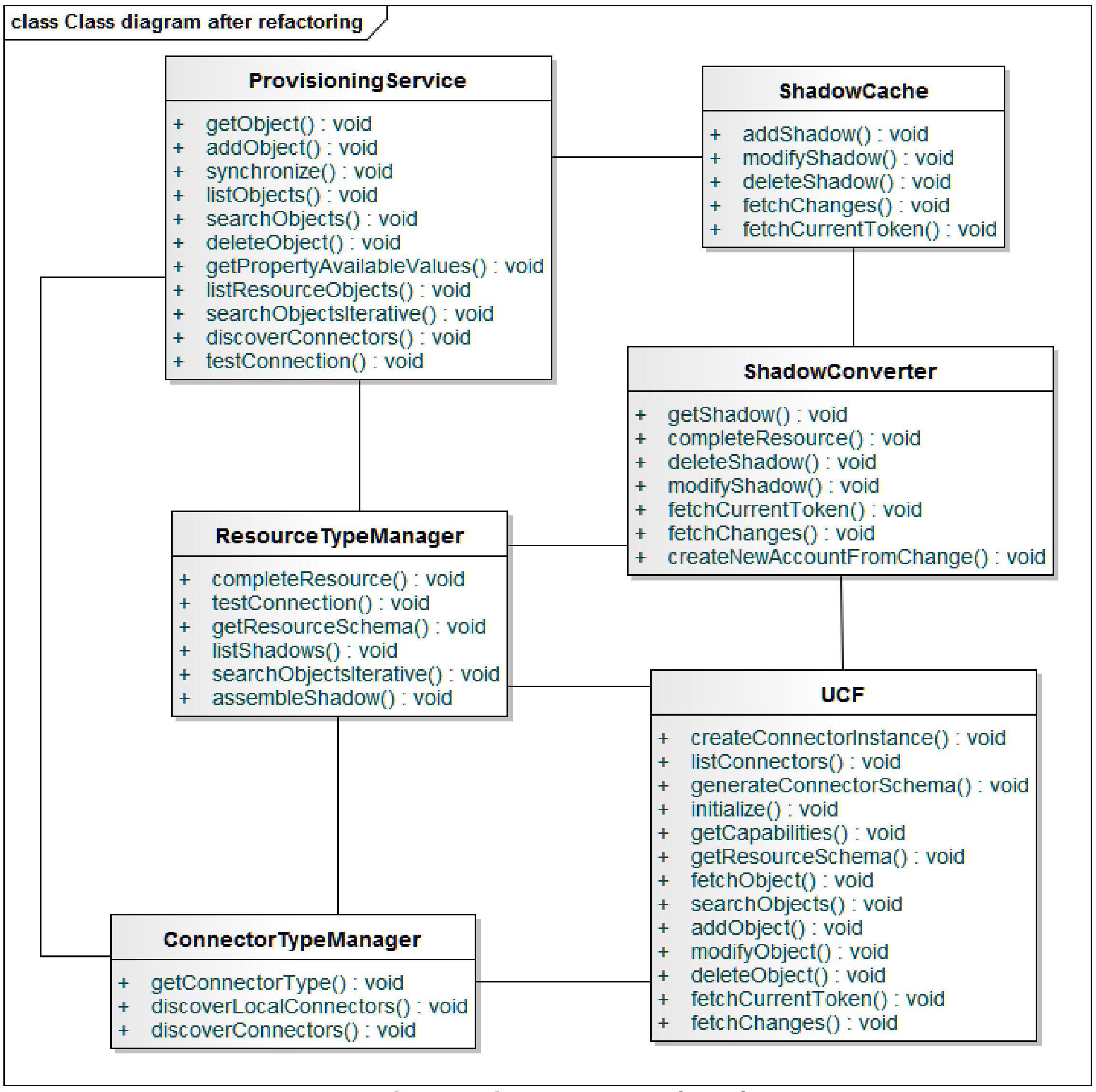


*Figure 21: The part of the data model related to the provisioning.*

# Appendix C – Solution Design Diagrams

## C.1 Preparing midPoint for Proposed Solution

Earlier, it was state, that the provisioning component seems as the suitable place where to apply the mechanism for consistency issues. Before applying these changes, the refactoring of the provisioning part is needed. Actually, there is one big class that is coupling with the propagating resource objects or entitlements to the external resources and also to local repository. This class is responsible also for transforming between shadow objects and resource objects. For the thesis purposes it is needed to divide this class to logical units, where each unit should be responsible for the certain action. The suggested refactoring is shown in figure 22.



class Class diagram after refactoring
ProvisioningService
+ getObject() : void
+ addObject() : void
+ synchronize() : void
+ listObjects() : void
+ searchObjects() : void
+ deleteObject() : void
+ getPropertyAvailableValues() : void
+ listResourceObjects() : void
+ searchObjectsIterative() : void
+ discoverConnectors() : void
+ testConnection() : void
ShadowCache
+ addShadow() : void
+ modifyShadow() : void
+ deleteShadow() : void
+ fetchChanges() : void
+ fetchCurrentToken() : void
ShadowConverter
+ getShadow() : void
+ completeResource() : void
+ deleteShadow() : void
+ modifyShadow() : void
+ fetchCurrentToken() : void
+ fetchChanges() : void
+ createNewAccountFromChange() : void
ResourceTypeManager
+ completeResource() : void
+ testConnection() : void
+ getResourceSchema() : void
+ listShadows() : void
+ searchObjectsIterative() : void
+ assembleShadow() : void
UCF
+ createConnectorInstance() : void
+ listConnectors() : void
+ generateConnectorSchema() : void
+ initialize() : void
+ getCapabilities() : void
+ getResourceSchema() : void
+ fetchObject() : void
+ searchObjects() : void
+ addObject() : void
+ modifyObject() : void
+ deleteObject() : void
+ fetchCurrentToken() : void
+ fetchChanges() : void
ConnectorTypeManager
+ getConnectorType() : void
+ discoverLocalConnectors() : void
+ discoverConnectors() : void

*Figure 22: Classes in the provisioning after refactoring.*

The class ShadowCache after refactoring contains only those methods by which inconsistencies should occur. It handles only shadow objects and it doesn't know about the resource objects. The responsibility for transforming the shadow objects to the resource objects and opposite has the class ShadowConverter. ShadowConverter also propagates the transformed resource objects to the external resources and the result returns to the ShadowCache.

ShadowChache takes full care for shadow objects, propagates them to the local midPoint repository. It calls ShadowConverter and contains the result of the running operation if it was successful or failed. Therefore, handling of the errors which should result to the inconsistencies of the object will be composed to this class (ShadowCache).

After refactoring of ShadowCache also the class ResourceTypeManager was formed. Its responsibility is to handle operations coupled with the resources. It takes care for example for generating resource schema or tests connection to the resource. The classes ConnectorTypeManager and ProvisioningService didn't change and they have original responsibility.

## *C.2 Extension of Account Object*

As it was stated earlier by the designing the reconciliation process (0 Reconciliation Process), there is a need to extend the shadow objects. Account shadow should contain also information that is needed for reconciliation process. Therefore the Schema is expanded with operation results, the number of attempts made to repair the inconsistencies problems and description of changes made to the account. These properties are optional and they are only used by recording the error state of the operation with the concrete account. The Schema extension is shown in the next example:

```
<xsd:complexType name="ResourceObjectShadowType">
 …
 <xsd:sequence>
 ...
    <xsd:element name="result" type="tns:OperationResultType”  minOccurs="0">
          <xsd:annotation>
                    <xsd:documentation>
                    Result describing if shadow was successfully processed, or not. If not, the errors should  be
saved.
                    </xsd:documentation>
          </xsd:annotation>
    </xsd:element>
    <xsd:element name="objectChange" type="tns:ObjectChangeType" minOccurs="0">
          <xsd:annotation>
                    <xsd:documentation>
                    Description of changes that happened to an resource object shadow.
                    </xsd:documentation>
          </xsd:annotation>
    </xsd:element>
    <xsd:element name="attemptNumber" type="xsd:int" minOccurs="0">
```

```
            <xsd:annotation>
                    <xsd:documentation>
                    Description of number of attempts made for the resolving account consistency issues.
                    </xsd:documentation>
            </xsd:annotation>
     </xsd:element>
  ...
  </xsd:sequence>
 ...
</xsd:complexType>
```

After extending the XML Schema the data model of the account shadow is as shown the figure 23.

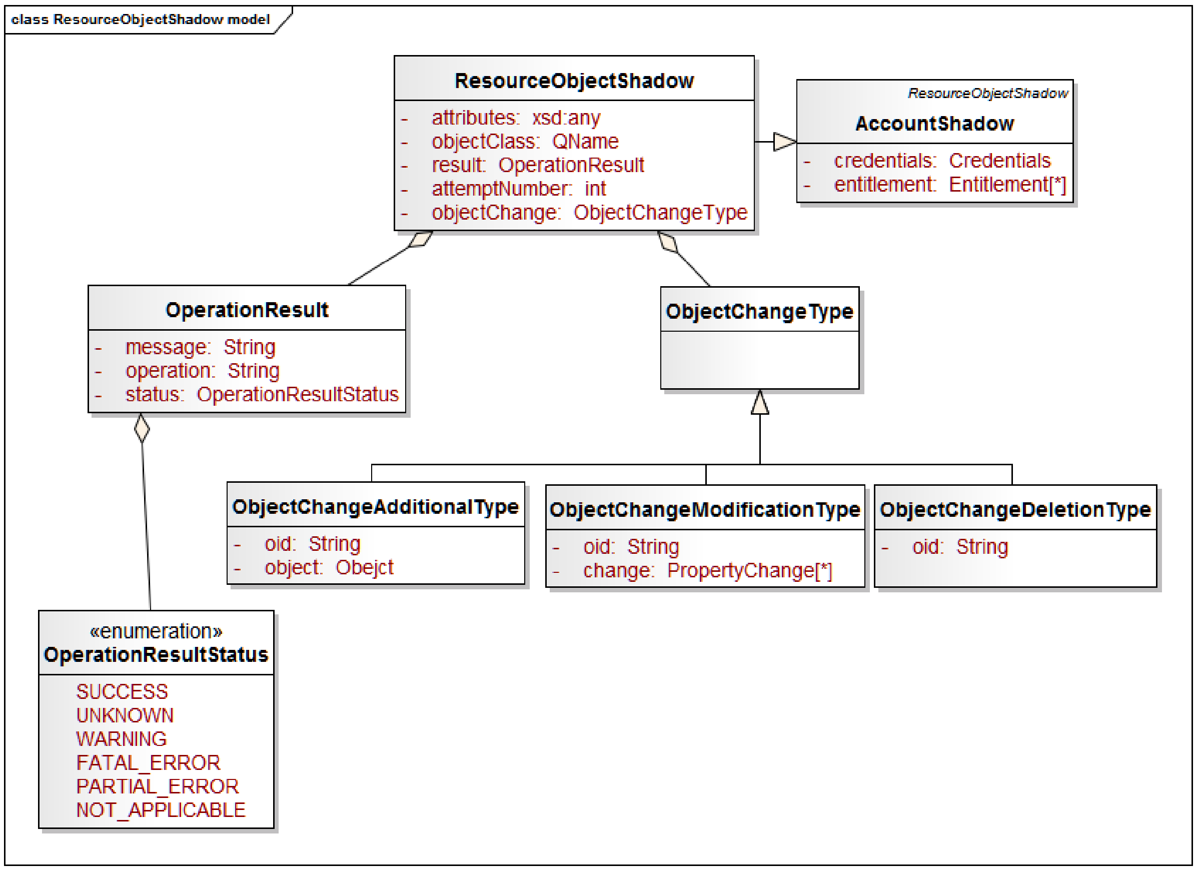


*Figure 23: ResourceObjectShadow after expanding the Schema.*

# Appendix D - Tests results

In next sections the individual results of the automated tests are provided. The next section's names describe the error which occurs and the error handler which is called to compensate the error.

*Table 7: Tests provided for the error ObjectNotFoundException and handler ObjectNotFoundHandler.*

| Action | Description | Expected result | Real result |
|---|---|---|---|
| Delete account | The end user tries to delete some of the user's (user1) account (account1), but account1 was not found on the resource. | Account1 is deleted from the midPint's repository and unlinked from the user1. | ✓ |
| Modify account | The end user tries to modify some of the user (user1) account's attributes (account1), but account1 was not found on the resource. User1 **contains** assignment that account1 should exist on the external resource. The attributes' modifications are defined in the changes1. | User1 is synchronized and according to assignment, the new account (account2) is created and linked to the user1. Account1 is deleted from the midPoint's repository and unlinked from the user1. The changes (changes1) are applied on the account2[8]. | ✓ |
| Modify account | The end user tries to modify some of the user (user1) account's attributes (account1), but account1 was not found on the resource. User1 **does not contain** assignment that account1 should exist on the external resource. The attributes' modifications are defined in the changes1. | User1 is synchronized, account1 id deleted from the midPoint's repository and unlinked from the user1. Changes1 are not applied and the end user gets message that changes cannot be applied, because object not exist. | ✓ |

[8] Account1 and account2 should be the same (they should have the same attributes). They differ in the midPoint's repository identificator, therefore there is stated that the changes are applied on the account2.

*Table 8: Tests provided for the error ObjectAlreadyExistException and handler ObjectAlreadyExistHandler*

| Action | Description | Expected result | Real result |
|---|---|---|---|
| Add account | The end user tries to add account (account1) to the user (user1) on the resource, but there exist account (account2) with the same identifier. Account2 does not have link with any user in the midPoint's repository and according to defined rules, account2 **belong** to the user1 | Found account (account2) is linked to the user (user1). The end user gets the error with the success status. The user1 has reference on the account2. | ✓ |
| Add account | The end user tries to add account (account1) to the user (user1) on the resource, but there exist account (account2) with the same identifier. Account2 does not have link with any user in the midPoint's repository and according to defined rules, account2 **does not belong** to the user1 | New user (user2) with link to the account2 is created in the midPoint's repository. Identifiers for account1 are recomputed and account1 is re-tried to add with the new identifiers. The result is that account1 is added and linked to the user1. | ✓ |
| Add account | The end user tries to add account (account1) to the user (user1) on the resource, but there exist account (account2) with the same identifier. Account2 is actually linked to other user (user2). | Identifiers for the account1 are recomputed and account1 is re-tried to add with the new identifiers. The result is that account1 is added and linked to the user1. | ✓ |
| Add account | The end user tries to add account (account1) to the user (user1) on the resource, but there exist account (account2) with the same identifier. According to defined rules, account2 should not exist on the resource and it is claimed as illegal. | Account2 is deleted from the resource and also from the midPoint's repository. Account1 is re-tried to add with the initial identifiers (no re-computation performed) | ✓ |

*Table 9: Tests provided for the error CommunicationException and the handler CommunicationExceptionHandler.*

| Action | Description | Expected results | Real result |
|---|---|---|---|
| Add account | End user tried to add account (account1) to the user (user1), but the resource where should be the account1 added is not reachable. | Account1 is stored to the midPoint's repository with additional information (failed operation type, result, attempt number, etc.) and it is also linked to the user1. | ✓ |
| Modify account | End user tries to modify account (account1) on the resource which is not reachable. Changes of the account are described in changes1. | Changes1 are stored to the account1 in the midPoint's repository. Also other additional information (failed operation type, result, etc.) are stored to the account1. | ✓ |
| Modify account | End user tries to modify account (account1) on the resource which is not reachable. Account1 was modified during the resource outage twice. The first changes (changes1) are stored with other additional information in the repository. New changes of the account are described in changes2. | Changes2 are compared with changes1 and if there is change of the same attribute, change2 value wins and is stored to the repository. If changes2 contains other changes than changes1, these changes are added into the midPoint's repository. Also other additional information (failed operation type, result, etc.) are stored to the account1. | ✓ |
| Modify account | End user tries to modify account (account1) on the resource which is not reachable. Account1 was actually created also when the resource was down, so it not created on the resource so far. Changes of the account1 are described in changes1. | Additional information stored with the account1 is updated (e.g. failed operation type). The old information is preserved and the information about change is added. Change1 is saved to the account1 in the midPoint's repository. | ✓ |
| Delete account | End user tries to delete account (account1) of the user (user1) on the resource which is not reachable. | Account1 is unlinked from the user1. Account1 is signed as a "dead" account in the repository. | ✓ |

*Table 10: Tests provided for reconciliation process.*

| Action | Description | Expected result | Real result |
|---|---|---|---|
| Add account | Account found by the reconciliation process should be added. The provisioning is called to add account on the external resource. | Account is added on the external resource. After added, account is cleaned and modified in the midPoint's repository. It means that account no more contains additional information. | ✓ |
| Modify account | Account found by the reconciliation process should be modified. The provisioning is called to modify account on the external resource. | Account is modified on the external resource. After modified, account is cleaned and modified in the midPoint's repository. It means that account no more contains additional information. | ✓ |
| Delete account | Account found by the reconciliation process should be deleted. The provisioning is called to delete account on the external resource. | Account is deleted on the external resource and also in the midPoint's repository. | ✓ |

# Appendix E - Glossary

| | |
|---|---|
| 2PC | two-phase commit protocol. It is used for transactions in the distributed systems. |
| ACID | properties used to describe transactions. It stand for Atomicity, Consistency, Isolation, Durability |
| BASE | The approach that allows temporary inconsistencies. It is opposite to the ACID transactions |
| BaseX | XML native database. There are saved XML files. |
| CAP Theorem | Theorem which shows that there is no possibility to guarantee consistency, availability and partition tolerance at the same time in the distributed systems. |
| ESB | stands for Enterprise Service Bus and provides communication between different systems using Web Services. |
| ICF | stands for Identity Connector framework. This framework is used to connect with the IDM solution to the different end systems |
| IDM | stands for Identity Management. |
| JAXB | stands for Java Api XML Binding. It is used for translation between Java objects and XML. |
| JUnit | is test framework used for automatically test operations of system. |
| LDAP | stands for Lightweight Directory Access Protocol and is used for working with directory services. |
| OpenDJ | is a directory service. |
| RBAC | stands for Role Base Access Control. It is used to define the access rights by defining a role of the user. |
| S3 | stands for Simple Storage Service. It is a protocol introduced by Amazon that allows temporary inconsistencies. |
| Saga | The name for long living transactions. |
| SOA | stands for Service Oriented Architecture. |
| Spring | application development framework. |
| SQS | stands for Simple Query Service. |
| SSO | stands for Single Sign On. It is a system that allows sharing of login. |
| UCF | stands for Unified Connector Framework and it is extension of ICF. It should solve some of the ICF known bugs. |

# Appendix F - Administration Manual

In this section the installation guide and the user guide is provided.

## F.1 Installation Guide

Installation of midPoint and the mechanism covering the thesis requires:

- Java SE Development Kit 6 – it is recommended at least update 28.
- Java Cryptography Extension Unlimited Strength Jurisdiction Policy Files 6.
- Apache Tomcat – it is developed and tested on the Apache Tomcat 6.
- Some SQL Database or XML Database BaseX.
- Some target system (e.g. OpenDJ which was used by testing the consistency mechanism).
- Build system Maven.
- Subversion.
- Source codes present on the attached CD.

The detailed installation tutorial for midPoint can be found on the midPoint’s official wiki page[9]. The step of getting midPoint from the repository should be skipped and instead of that the attached source code should be used. Or if you want to install the current version (under development, version 2.0) of the midPoint, you can get it from the subversion repository[10] and then follow the tutorial steps. The mechanism implemented for the thesis purposes is the part of the currently developed version.

## F.2 User Guide

User guide is available at midPoint official wiki page[11]. For running midPoint the application server Apache Tomcat should be started and the midPoint should be deployed on the server. Also there should exist some of the target system, e.g OpenDJ and it should be also started. Performance of individual steps for creating user and account are intuitive. If you would like add a new account, you must first create user and import the resource. The samples of different resources should be found in midPoint\samples. After the resource is imported and the user is created, it is good to test connection to the end resource. This can be done through the midPoint web interface going to the Resource -> select resource -> test connection. If the connection is open, the new account should be

[9] http://wiki.evolveum.com/display/midPoint/Installing+midPoint+from+Source+Code+Release+-+1.10

[10] https://svn.evolveum.com/midpoint/trunk

[11] http://wiki.evolveum.com/display/midPoint/First+Steps

added. Operations coupled with the account are in midPoint web interface situated in the user details page. Simply select the user and the page will be displayed. Then click the edit button and the possibilities to add account, delete account or modify account will be displayed. The midPoint web interface with some explanations is shown in the figure 24.

## *F.3 Testing Guide*

This section describes the tutorial for automatic tests and also for the manual testing. The figure 24 may be helpful also for the testing tutorial.

### F.3.1 Automatic tests

The next steps describe how to run the automated tests.

1. Build midPoint's source code (current version from the repository or attached on the CD).
2. Start command line.
3. Go to the directory, where the source codes are present.

   e.g. *cd D:\midPoint\trunk*

4. Run the following line from your command line to build source code:

   *mvn –P default clean install –Dmaven.test.skip=true*

   It is a command for the maven build system to build source code. The –P switch is used to set the profile. If default is set, it means it will build without aspects. The switch –Dmaven.test.skip determines if the tests will run. If set to true, it will be skipped.

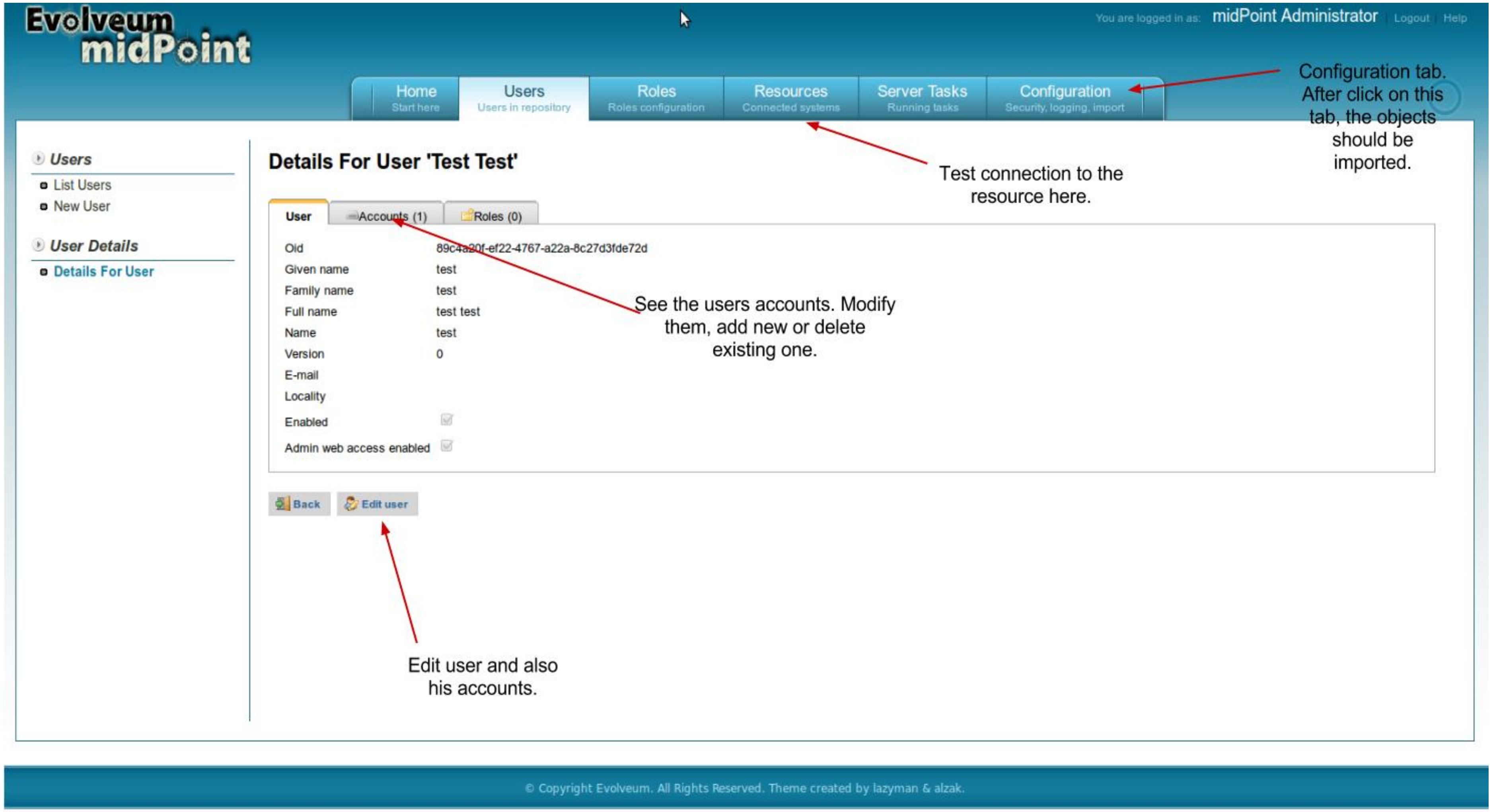
5. Go to the consistency-test directory.

   *cd testing\consistency-mechanism*

6. Run the following line from your command line for launching tests:

   *mvn –P default clean test*

7. After the tests finish, there is displayed the result of the tests.


Evolveum midPoint

You are logged in as: midPoint Administrator | Logout | Help

Home Start here | Users Users in repository | Roles Roles configuration | Resources Connected systems | Server Tasks Running tasks | Configuration Security, logging, import

Configuration tab. After click on this tab, the objects should be imported.

Test connection to the resource here.

Users
- List Users
- New User

User Details
- Details For User

Details For User 'Test Test'

User | Accounts (1) | Roles (0)

| | |
|---|---|
| Oid | 89c4a20f-ef22-4767-a22a-8c27d3fde72d |
| Given name | test |
| Family name | test |
| Full name | test test |
| Name | test |
| Version | 0 |
| E-mail | |
| Locality | |
| Enabled | ✓ |
| Admin web access enabled | ✓ |

See the users accounts. Modify them, add new or delete existing one.

Back | Edit user

Edit user and also his accounts.

© Copyright Evolveum. All Rights Reserved. Theme created by lazyman & alzak.

*Figure 24: Screenshot of midpoint (user details page).*

## F.3.2 Manual testing

For manual testing the midPoint solution must be first installed. Follow the installation guide introduced in the F.1 Installation Guide. If the midPoint is installed, the first steps describing how the users, account and other object should be created are on the midPoint's official wiki page[12]. Individual situations are described below.

**Object not found – modify account that is not assigned**

1. Deploy midPoint on the Apache Tomcat.
2. Log in to the midPoint using username: administrator and password:secret.
3. Import XML file with the resource configuration (sampes/resource-opendj-advanced-synchronization.xml from the attached CD, in midPoint: configuration -> import -> select the file -> press the button import).
4. Start OpenDJ.
5. Test connection for the OpenDJ through the midPoint (resources -> openDj -> test connection).
6. Create user through the midPoint (users -> create user)
7. Add account for the user through the midPoint (select user -> edit -> accounts -> add account
   -> submit). Attributes have not to be filled, they will be computed according to outbound expressions.
8. Ensure the account was created on the resource and corresponding account shadow in the midPoint's repostiory (e.g. for OpenDJ – run control-panel.bat for windows. Loggin with default name and password: secret, click on the manage entries, expand dc=example, dc=com directory and check if the account is present).
9. Modify some of the account's attributes (select user -> edit -> add, remove or change some of the attribute value but do not press the submit button).
10. Delete account from the resource.
11. Press the submit button in the midPoint edit user page.
12. Check the message after the operation finished.

**Object not found – modify account that is assigned**

1. Deploy midPoint on the Apache Tomcat.
2. Log in to the midPoint using username: administrator and password:secret.
3. Import XML file with the resource configuration (sampes/resource-opendj-advanced-synchronization.xml from the attached CD).
4. Start OpenDJ.
5. Test connection for the OpenDJ through the midPoint (resources -> openDj -> test

[12] http://wiki.evolveum.com/display/midPoint/First+Steps

connection).

6. Import the role through the midPoint (roles -> created role -> paste the role definition from the file samples/role-example.xml -> press the button create)
7. Create user through the midPoint (users -> create user)
8. Assign the role for the user (select user -> edit -> roles -> add role -> select role -> press the button submit)
9. Ensure the account was created on the resource and corresponding account shadow in the midPoint's repostiory (e.g. for OpenDJ – run control-panel.bat for windows. Loggin with default name and password: secret, click on the manage entries, expand dc=example, dc=com directory and check if the account is present).
10. Modify some of the account's attributes (select user -> edit -> add, remove or change some of the attribute value but do not press the submit button).
11. Delete account from the resource (manage entries -> dc=example, dc=com -> select account -> right click -> delete entry).
12. Press the submit button in the midPoint edit user page.
13. Check the message after the operation finished.

**Object not found – delete account**

1. Deploy midPoint on the Apache Tomcat.
2. Log in to the midPoint using username: administrator and password:secret.
3. Import XML file with the resource configuration (sampes/resource-opendj-advanced-synchronization.xml from the attached CD, in midPoint: configuration -> import -> select the file -> press the button import).
4. Start OpenDJ.
5. Test connection for the OpenDJ through the midPoint (resources -> openDj -> test connection).
6. Create user through the midPoint (users -> create user)
7. Add account for the user through the midPoint (select user -> edit -> accounts -> add account
   -> submit). Attributes have not to be filled, they will be computed according to outbound expressions.
8. Ensure the account was created on the resource and corresponding account shadow in the midPoint's repostiory (e.g. for OpenDJ – run control-panel.bat for windows. Loggin with default name and password: secret, click on the manage entries, expand dc=example, dc=com directory and check if the account is present).
9. Delete account from the resource (manage entries -> dc=example, dc=com -> select account -> right click -> delete entry).
10. Delete account from the user through the midPoint (select user -> edit -> delete account -> press the submit button).

11. Check the message after operation finished.

**Object already exists – add account, found account should be linked**

1. Deploy midPoint on the Apache Tomcat.
2. Log in to the midPoint using username: administrator and password:secret.
3. Import XML file with the resource configuration (sampes/resource-opendj-advanced-synchronization.xml from the attached CD, in midPoint: configuration -> import -> select the file -> press the button import).
4. Start OpenDJ.
5. Test connection for the OpenDJ through the midPoint (resources -> openDj -> test connection).
6. Add account on the resource directly (manage entries -> dc=example, dc=com -> new entry -> fill the attributes -> create)
7. Create user through the midPoint (users -> create user). Use the same name for the user as you used for the account on the resource.
8. Add new account to the user through the midPoint (select user -> edit -> accounts -> add account -> press the submit button).
9. Check if the previously created account on the resource was linked to this user (check resource if there are not additional accounts created, check the user added accounts: select user -> accounts).

**Object already exist – add account, found account is linked to the another user**

1. Deploy midPoint on the Apache Tomcat.
2. Log in to the midPoint using username: administrator and password:secret.
3. Import XML file with the resource configuration (sampes/resource-opendj-advanced-synchronization.xml from the attached CD, in midPoint: configuration -> import -> select the file -> press the button import).
4. Start OpenDJ.
5. Test connection for the OpenDJ through the midPoint (resources -> openDj -> test connection).
6. Create user1 through the midPoint (users -> create user (user1)).
7. Add account to the user1 through the midPoint(select user -> edit -> accounts -> add account -> press the submit button). Fill the account name attribute for example with the value uid=test,ou=people,dc=example,dc=com.
8. Create user2 through the midPoint (users -> create user). As the name of the user set “test”.
9. Add account to the user2 through the midPoint (select user -> edit -> accounts -> add account -> press the submit button). Do not fill the account’s attributes.
10. Check if the user2 has account and check the account identifier.

**Communication error – add account**

1. Deploy midPoint on the Apache Tomcat.
2. Log in to the midPoint using username: administrator and password:secret.
3. Import XML file with the resource configuration (sampes/resource-opendj-advanced-synchronization.xml from the attached CD, in midPoint: configuration -> import -> select the file -> press the button import).
4. Start OpenDJ.
5. Test connection for the OpenDJ through the midPoint (resources -> openDj -> test connection).
6. Create user through the midPoint (users -> create user)
7. Add account for the user through the midPoint (select user -> edit -> accounts -> add account -> do not press the submit button).
8. Stop the OpenDJ (in the control panel, press the stop button).
9. Submit the unfinished account.
10. Check if the user has a new link created.
11. Check the midPoint repository if there is an account which contains the failed operation type, operation result and other additional information

**Communication error – modify account**

1. Deploy midPoint on the Apache Tomcat.
2. Log in to the midPoint using username: administrator and password:secret.
3. Import XML file with the resource configuration (sampes/resource-opendj-advanced-synchronization.xml from the attached CD, in midPoint: configuration -> import -> select the file -> press the button import).
4. Start OpenDJ.
5. Test connection for the OpenDJ through the midPoint (resources -> openDj -> test connection).
6. Create user through the midPoint (users -> create user)
7. Add account for the user through the midPoint (select user -> edit -> accounts -> add account -> press the submit button).
8. Modify some of the account's attributes (select user -> edit -> add, remove or change some of the attribute value but do not press the submit button).
9. Stop the OpenDJ (in the control panel, press the stop button).
10. Submit the unfinished account changes.
11. Check the midPoint repository if there is an account which contains the failed operation type, operation result, changes description and other additional information.

**Communication error – delete account**

1. Deploy midPoint on the Apache Tomcat.
2. Log in to the midPoint using username: administrator and password:secret.

3. Import XML file with the resource configuration (sampes/resource-opendj-advanced-synchronization.xml from the attached CD, in midPoint: configuration -> import -> select the file -> press the button import).
4. Start OpenDJ.
5. Test connection for the OpenDJ through the midPoint (resources -> openDj -> test connection).
6. Create user through the midPoint (users -> create user)
7. Add account for the user through the midPoint (select user -> edit -> accounts -> add account -> press the submit button).
8. Delete account through the midPoint (select user -> edit -> accounts -> delete account but do not press the submit button).
9. Stop the OpenDJ (in the control panel, press the stop button).
10. Submit the unfinished account.
11. Check if the user no more has a link to the account.
12. Check the midPoint repository if there is an account which contains the failed operation type, operation result and other additional information.

**Reconciliation**

1. Deploy midPoint on the Apache Tomcat.
2. Log in to the midPoint using username: administrator and password:secret.
3. Import XML file with the resource configuration (sampes/resource-opendj-advanced-synchronization.xml from the attached CD, in midPoint: configuration -> import -> select the file -> press the button import).
4. Start OpenDJ.
5. Test connection for the OpenDJ through the midPoint (resources -> openDj -> test connection).
6. Import the reconciliation task (configuration -> import -> sampes/recon-task-opendj.xml -> press the import button).
7. Wait a minute.
8. Check the midPoint’s repository for previously failed accounts. They should be cleaned.

# Appendix G - Paper accepted for IIT.SRC

# Consistency in the Identity Management

Katarína VALALIKOVÁ*

*Slovak University of Technology in Bratislava*
*Faculty of Informatics and Information Technologies*
*Ilkovičova 3, 842 16 Bratislava, Slovakia*
k.valalikova@gmail.com

**Abstract.** Identity management solutions are integrative solutions that automate processes associated with the users and theirs life-cycle. The results of the identity management solutions can be thought of as (loosely coupled) distributed systems. Many of the end resources managed by the identity management systems do not support transactions or other traditional consistency mechanisms therefore there is a considerable risk of inconsistencies. In this paper we introduce the mechanism which will be able to minimize the formation of inconsistencies and if they ever happen, this mechanism will be trying to resolve them and bring the system back to the consistent state.

## 1 Introduction

Identity management can be defined as a convergence of technologies and business processes [8]. It is integrative solution that usually consists of different systems and techniques. The main goal of identity management is to handle a lot of identities and their life-cycle including creation, usage, updating and revocation of the identity [1]. Identities have different roles and different permissions to access specified resources. There is a need to have different identities to work with the same system, or to have the same identity to work with different systems [8].

It is important to solve the inconsistency problems for many reasons. For example, the identity management solution interacts with various systems and information about user's identity is stored in more than one database. Without any reliable consistency mechanism the databases may diverge and it may not be clear which data record should be used. Another reason why it is needed to solve the problem with inconsistencies may be security. The identity management solutions are security-sensitive systems because they manage accesses to other systems. Consistency of security policy is important for maintaining good security level and also for being able to monitor overall security of the organization. For instance, potential attacker can intentionally cause inconsistency and escape the "security police".

The paper is organized as follows. The section 2 gives the brief view on the related work in the area of the transaction mechanisms. Next section, section 3, deals with the problems which must be solved for ensuring consistency in the identity management and also in this section the consistency mechanism is introduced. The section 4 evaluates the proposed mechanism and in the section 5 is the conclusion summarizing the proposed mechanism and the future work.

* Master degree study programme in field: Software Engineering
Supervisor: Dr. Radovan Semančík, Institute of Informatics and Software Engineering, Faculty of Informatics and Information Technologies STU in Bratislava

## 2 Related work

Consistency in the database systems is guaranteed by transactions described using ACID properties (Atomicity, Consistency, Isolation, Durability) [9]. Transactions in the distributed systems are often realized using two-phase commit protocol that is responsible for coordination of all participants in the transaction and for decision if the transaction will be committed or rolled back. Before all transaction are either committed or aborted, the data in the database are locked to enforce concurrency control [10]. The paper [4] describes an approach where a long running transaction called *saga* is divided into shorter sub-transactions where each sub-transaction has its own compensation mechanism. If transaction fails, compensation mechanism is called to undo this transaction. Success of the all sub-transactions implies success of the whole transaction.

According to CAP theorem [3, 5] it is impossible to guarantee consistency, availability and partition tolerance at the same time in distributed systems. The paper [5] provides a proof of the theorem using asynchronous network model. With respect to the CAP theorem author in the [7] introduces approach called BASE (Basically available, Soft state and Eventual consistent system) where he suggests to establish persistence message queue and to divide the database tables across functional groups (table for users, transactions, messages, etc.). Each message is peeked from the queue and processed. If operation is successful, the message is removed from the queue, otherwise it is re-tried [7]. The eventual consistency is also provided by the Simple Storage Service (S3) introduced by Amazon [2].

However, many of the end resources managed by the identity management do not support transactions. Therefore the standard transactions, two-phase commit protocol or other traditional consistency mechanism are not suitable for ensuring consistency in the identity management and the other mechanism must be found. In addition, many of the operations in the identity management can take a long time so there cannot be used the standard lock and log mechanism. There is also problem with the Sagas. The author in [6] explains that Sagas fail to cover all cases where compensation is needed. Following these facts, the best approach to ensure the consistency in the identity management seems to be the concept of eventual consistency. Therefore we decide to use this model to design the mechanism.

## 3 Consistency mechanism for non-transactional systems

The goal of this paper is to find an appropriate way to solve the consistency issues in identity management systems. The identity management system must be able to recover from unexpected errors and to continue to work without limiting the users. It is unacceptable to allow identity management to be in the inconsistent state for a long time because this could result to the malfunction of the system.

Identity management systems provide automation of the processes related to the users and their life-cycle in the enterprise, from hiring new employee through changing his position to firing employees. Each employee usually has multiple accounts in the various systems to be able to perform his work properly. Therefore, there are a lot of external resources which need to communicate with the identity management systems. External resources contain information about the employees and their access rights, one employee should have accounts in the different resources and may also have more than one account in the same resource.

Accounts are created in different ways, e.g. using central identity management system, by synchronization of changes on external resources, or by adding the user to the role which defines that an account should be created, etc. Unexpected situations and errors may happen during the user management processes, e.g. the account may not be created, exceptions may be thrown, etc. Ultimately, this may lead to the inconsistency of the record. According to the way the inconsistencies originate we can divide them into the following categories:

- Resource failures – this group describes failures that happened on the external resource by propagating changes that were made by end user using identity manager. For example, adding

account through the identity management to the external resource which is not reachable.
- Synchronization failures – this group describes failures that happened by synchronization. Changes on the external resource was detected and propagated to other external resources and also to the identity manager but some crash occurred.
- Dependencies – this group describes inconsistencies that should happened by creating account that may have dependencies to other accounts. For example, creation of the account in the application depending on the account in the operation system.
- Groups – this group describes failures that happened when some operation with the group was made. For example, creation of account and adding it to the group are in the LDAP two different operations.
- Role Assignment – this group describes inconsistencies that occurred while working with roles. For example, the role is defined to have four different accounts, but only two of them are successfully created and the question is what to do with such a role.

## 3.1 Proposed mechanism

Proposed solution follows the model of the eventual consistency which means that the system does not guarantee that data will be consistent all the time. Instead, the temporary inconsistencies are allowed and the attention is made for the mechanism which solves the inconsistencies and eventually brings the data to the consistent state.

The main reason why we decided to use the weak consistency model results from the CAP theorem, because for such systems like identity management solutions are, it is required to guarantee high availability and so you can read and write to the system all the time. Every request to the system must have appropriate response even if failures occurs (e.g. one of the node is down). It does not matter if the operation was successful, but it must be terminated and the result returned to the user. Even, if a message sent from one node to another is lost, the system must continue to operate. Mechanism proposed in this paper is based on the three base concepts:

- CAP theorem, where the availability and partition-tolerance is chosen and the consistency is weakened.
- Relative change model - this means that in the case when the object was changed, we do not assume absolute state of the object (all its attributes) before the change and after the change, but we only use the attributes that have been really changed.
- Compensations for the unsuccessful operations. Each operation can end successfully or unsuccessfully. If the operation ends with an error, we first decide if this error is processable. If yes, then the compensation mechanism is called to resolve the problem, otherwise we can do nothing.

The mechanism is proposed to minimize the formation of the inconsistencies and if they ever happened, it should reasonably react and bring the system to the consistent state. The mechanism was designed with respect to the one of the identified groups of problems – resource failures. It consists of two parts. The first part tries to handle the unexpected error that occurred and the second part, called reconciliation, is used to compare the state of the object stored in the repository and in the end resource.

It has to be known if the error that occurred is processable or not in the first part. If the error is processable, there are specified compensation mechanisms as the reaction to the error. Each error has its own compensation mechanism. The simple example of the compensation mechanism is, when we try to add the account to the end resource which is actually unreachable. In this case, we want to add this account later, but without the need of the user's assistance. Therefore the whole account object including all of its attributes is stored to the IDM system database and it is also marked as unsuccessfully added object to be able to process it later by reconciliation.

The more complex example can be a scenario, when we need to process the error immediately. This is a group of errors that we know either process immediately and eliminate the error state or we cannot do anything and the error must be repaired by the user, e.g. adding account that actually exist on the resource shown in Figure 1.

If the error is not processable it means, that we do not know how to implement the compensation to the error. Such an error can be also considered as fatal, and then the user help is needed for its reparation. As the example we can mention the issue where no definition for the required attribute was specified. The attribute was not specified, and the objects cannot be added to the end resource.

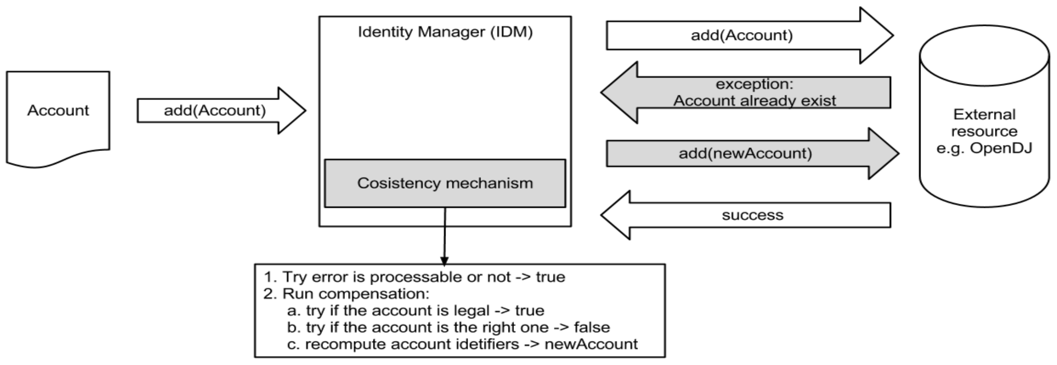

Account
add(Account)
Identity Manager (IDM)
Cosistency mechanism
add(Account)
exception: Account already exist
add(newAccount)
success
External resource e.g. OpenDJ
1. Try error is processable or not -> true
2. Run compensation:
a. try if the account is legal -> true
b. try if the account is the right one -> false
c. recompute account idetifiers -> newAccount

*Figure 1 The first part of the mechanism (compensation for the situation where the account already exist on the resource).*

Now, when the first step is done, it is needed to specify the policies for reconciliation process according to which the unsuccessful created accounts will be picked from the repository. This brings the idea that storing only account attributes is not enough. If there is no additional information to such account, it is quite impossible to declare it as unsuccessfully handled and it can be considerably dificult to find it by the reconciliation process and process it to resolve the inconsistency. Therefore, it is needed to expand the account object with some additional information which provides the detailed view about what and why went wrong. These are:

- all the account attributes when operation fails (e.g. creation of the account failed because of the end resource is unreachable and we want to try to add this account again later without he user's assistance),
- description of properties changed (e.g. the account was modified, but there was some error which avoided for applying the changes),
- some information about the operation and the type of error which occurred (e.g. the name of the operation, the status, is the operation was successful or not, the error which occurred),
- the number of the attempts made for re-try of operation.

Now, the account in the repository contains also information about operations and theirs results. The reconciliation process is used to scan the repository for the failed accounts and to try to run the failed operation again. The scanning is done with filter on the failed accounts and the number of attempts made for re-trying of operation. Found accounts are processed using additional attributes that describe the operation and its inputs to run again. We use the number of attempts because it makes no sense to retry the operation endlessly.

The reconciliation process ends either successfully or it can also fail. It will be implemented in the way to not limit the end user for his activity. After defined number of attempts it will not be interesting in the solving the inconsistencies. Figure 2 illustrates the reconciliation process.

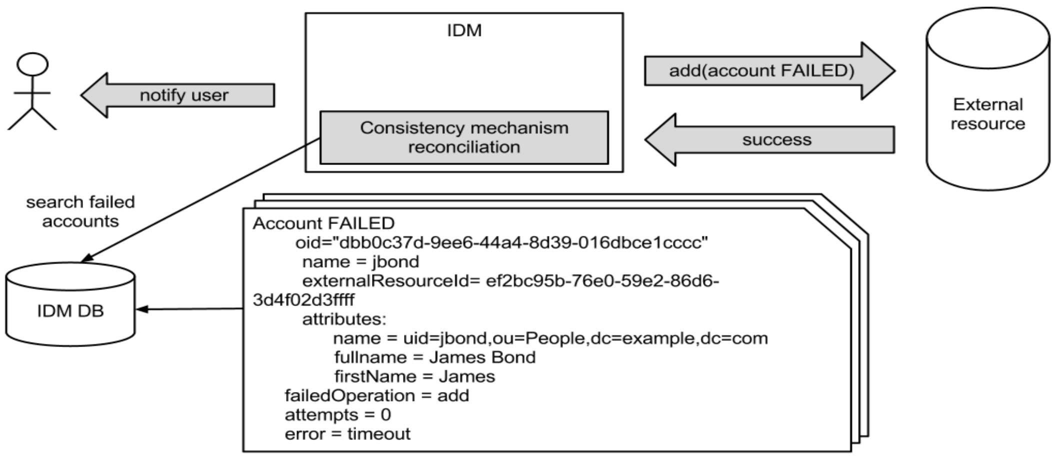


*Figure 2 Reconciliation process.*

## 4 Evaluations and Achievements

The theoretical proposal of the consistency mechanism was practically proven. The proof of concept was made on the existing identity management solution called midPoint[13]. MidPoint is open-source identity management solution and the main reason to choose it was the fact, that we are members of the development team so we have the practical experiences with the system.

The mechanism implementation consists of all parts proposed in the paper. There were identified concrete situations which can lead to the inconsistencies. To each situation we first decided if it is possible to solve the resulting problem and then we divided the situation into the processable and unprocessable errors. Each processable error has its compensation mechanism written according to its nature. If the compensation mechanism is not enough to resolve the issue, there is also implementation for the reconciliation process.

The implementation was tested manually by simulating the identified situations. We observed that this mechanism was able to properly recognize the situation and reasonably react to it. Where the compensation mechanism was not enough, the reconciliation process was used to additionally eliminate the inconsistencies in the system. The tests made on the prototype implementation are shown in the Table 1.

*Table 1 Tests provided on the prototype implementation of the consistency mechanism.*

| Situation | Error | Reaction |
|---|---|---|
| Add, Modify, Delete account | Connection problem | Account was saved to the repository. Reconciliation process finds this account and tries again to add it. |
| Add account | Already exist on the external resource | If the found account belongs to the specified user, the account was linked to the user, otherwise new account identifiers were generated. If the found account was illegal, it was deleted. |
| Modify account | Not found on the resource | If the account should exist, it was created and the modifications were applied, otherwise the modifications were discarded. |
| Delete account | Not found on the resource | The result of this reaction is that the account was deleted from the repository and it was also unlinked from the user. |

## 5 Conclusions

In this paper we introduced the mechanism for ensuring the consistency in the systems where the transactions

[13] http://evolveum.com/midpoint.php

are not supported. We proposed the mechanism with respect to the known mechanisms that was also analysed in the introduction of this paper. The mechanism is based on the relative change model, model of compensations and the CAP theorem according to which distributed systems cannot satisfy the consistency, availability and partition tolerance at the same time. Therefore we decided to weak the consistency and guarantee only the eventual consistency.

It means, we do not guarantee that after every operation the data are consistent. Instead, we allow temporary inconsistencies and we try to solve them and eventually bring the system to the consistent state. The implementation was made on the existing identity management system called midPoint. The future work can be concentrated on the other identified group of problems stated in the paper.

## References

[1] Bertino, E., Takahashi, K.: *Identity Management: Concepts, Techniques and Systems*. Artech House Publisher, Norwook MA, (2010).

[2] Brantner, B., et al.: Building a database on S3. *In Proceedings of the 2008 ACM SIGMOD, international conference on Management of data* (SIGMOD '08). ACM, New York, NY, USA, (2008), pp. 251-264.

[3] Brewer, E.: A certain freedom: thoughts on the CAP theorem. *In Proceeding of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing* (PODC '10). ACM, New York, NY, USA, (2010), pp. 335-335.

[4] Garcia-Molina, H., Salem, K.: Sagas. *In Proceedings of the 1987 ACM SIGMOD international conference on Management of data* (SIGMOD '87), (1987), pp. 249-259.

[5] Gilbert, D., Lynch, N.: *Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services*. SIGACT News 33, 2 (June 2002), pp. 51-59.

[6] Greenfield, P. et al.: Compensation is Not Enough. *In Proceedings of the 7th International Conference on Enterprise Distributed Object Computing* (EDOC '03). IEEE Computer Society, Washington, DC, USA, (2003), pp. 232-240.

[7] Pritchett, D.: BASE: *An Acid Alternative*. Queue 6, 3 (May 2008), pp. 48-55.

[8] Slone, S. and the open group identity management work area: *Identity Management*. (2004), pp. 1-109.

[9] Wang, T. et al.*: A survey on the history of transaction management: from flat to grid transactions.* Distributed Parallel Databases 23, 3 (June 2008), pp. 235-270.

[10] Wolfson, O.: *The overhead of locking (and commit) protocols in distributed databases*. ACM Trans. Database Systems 12, 3 (September 1987), pp. 453-471.

# Appendix H - Source Code (medium)

Attached CD contains:

- xvalalikova_diploma_thesis.pdf – the diploma thesis
- midPoint – the directory with the source code and samples. It is structured as follows:
    - build-system
    - gui
    - icf-connectors
    - ide
    - infra – the extension of schema is situated here. It can be found in the subdirectory schema -> infra\schema\src\main\resources\xml\ns\public\common\common-1.xsd.
    - legal
    - model – reconciliation and the part of the object already exists compensation can be found here in the subdirectory model-impl. For reconciliation see the file src\main\java\com\evolveum\midpoint\model\sync\ReconciliationTaskHandler.java
      For the part of the object already exists compensation see the file src\main\java\com\evolveum\midpoint\model\controller\ModelController.java
    - provisioning
        - provisioning-api
        - provisioning-impl -> in the provisioning-impl component is placed the consistency mechanism. It can be found in the packages: com.evolveum.midpoint.provisioning.consistency.api and com.evolveum.midpoint.provisioning.consistency.api. Some of the code needed for the mechanism is also placed in the ShadowCache class situated in the package com.evolveum.midpoint.provisioning.impl.
    - repo
    - samples – samples of the XML objects. They should be used for testing midPoint and the consistency mechanism.
    - Testing – in the subdirectory consistency-mechanism are placed automatic tests for consistency mechanism.
    - Tools
    - INSALL, pom.xml, README, RELEASE-NOTES
- README.TXT – which contains description of attached medium.

# Appendix I - Resume

Systémy na správu podnikových identít umožňujú automatizáciu procesov súvisiacich s používateľmi a celým ich životným cyklom, od vytvorenia, cez aktualizáciu až po vymazanie používateľa. Sú to integračné riešenia, pri ktorých centrálny systém na správu podnikových identít (Identity Manager) komunikuje z rôznymi externými systémami. Záznamy o identitách sú riadené z centrálneho systému (Identity manager) a sú propagované do riadených systémov. Keďže sa takto vytvára (voľne viazaný) distribuovaný systém a väčšina cieľových systémov nepodporuje distribuované transakcie, môžu nastávať nekonzistencie.

Nekonzistenciu údajov je potrebné riešiť z viacerých dôvodov. Príkladom môže byť situácie, keď sa zmeny vykonané na identite uložia v jednej databáze a v druhej nie. Vtedy máme k dispozícií dve verzie údajov a nevieme s určitosťou povedať, ktorá verzia je platná, údaje vo verziách sa líšia a môže to spôsobiť viaceré problémy.

Systémy na správu podnikových identít sú bezpečnostne citlivé lebo sa pracuje s prístupmi do systémov. Konzistencia bezpečnostnej politiky je dôležitá pre udržanie dobrej úrovne bezpečnosti a najmä pre možnosť bezpečnosť organizácie monitorovať. Napr. potencionálny útočník môže cielene vyvolať nekonzistenciu a tak uniknúť pozornosti bezpečnostnej „polície“.

V súčasnosti existuje množstvo existujúcich systémov na správu podnikových identít. V práci boli identifikované niektoré z nich ako napríklad OpenIDM [15], OpenIAM [12], OpenPTK [16] alebo midPoint [10]. Pri analyzovaní týchto riešení však neboli zistené žiadne zmienky o riešení konzistencie. Dôvodom je hlavne fakt, že množstvo koncových systémov, s ktorými centrálny systém (Identity Mnager) komunikuje, nepodporujú transakcie. Navyše, systémy na správu podnikových identít využívajú na komunikáciu s externými systémami konektory, ktoré takisto nepodporujú transakcie.

## *I.1 Súvisiace práce*

V práci boli analyzované viaceré existujúce techniky na podporu konzistencie údajov. Medzi najznámejšie patria transakcie. Transakcie sú bežne používate v databázových systémoch. Môžu byť definované ako transformácia z jedného stavu na iný s tým, že je zachovaná atomicita, konzistencia a trvácnosť. Inak povedané, na opis transakcii sa často používajú vlastnosti ACID: A – atomicita, C – konzistencia, I – izolácia, D – trvácnosť [8]. V distribuovaných systémoch sú transakcie realizované využitím dvojfázového odovzdávacieho protokolu (z angl. two-phase commit protocol). Tento protokol pozostáva z dvoch fáz, hlasovacej a rozhodovacej. Hlasovacia fáza ma za úlohu zozbierať hlasy od jednotlivých zúčastnených strán a v rozhodovacej fáze sa potom rozhodne, či bude transakcia schválená alebo vrátená spat [25].

Ďalším prístupom, ktorý bol v práci analyzovaný je mechanizmus pre dlho trvajúce transakcie. Autor tieto dlho trvajúce procesy nazýva Saga. Autor v [6] navrhuje rozdeliť Saga na kratšie podtransakcie s tým, že každá podtransakcia má definovaný vlastný kompenzačný mechanizmus. V prípade, že nastane chyba počas transakcie, zavolá sa kompenzačný mechanizmus, ktorého úlohou je vrátiť dáta do stavu pred transakciou. Neznamená to však, že dáta budú v stave ako boli pred transakciou, pretože súčasne môžu prebiehať i iné transakcie, ktoré dáta zmenili. Úlohou kompenzačného mechanizmu je vrátiť zmeny len týkajúce sa danej transakcie.

V rámci analýzy bola uvedená aj CAP teoréma a niektoré mechanizmy vychádzajúce z nej. CAP teoréma hovorí, že v distribuovaných systémoch nie je možné zaručiť zároveň konzistenciu, dostupnosť a odolnosť voči výpadkom uzlov. Autor v [7] uvádza dôkaz využívajúc asynchrónny sieťový model, že je možné vybrať len dvojicu z týchto záruk, a teda dostupnosť a odolnosť voči výpadkom alebo konzistenciu dát a odolnosť voči výpadkom. Dvojica dostupnosť a konzistencia opisuje tradičné systémy na riadenie bázy dát. Uprednostnenie konzistencie či dostupnosti na úkor toho druhého závisí na charakteristikách systému.

Dostupnosť pred konzistenciou uprednostňuje prístup nazývaný BASE (z angl. Basically Available, Soft State and Eventually consistent). Tento prístup dovoľujúci dočasné nekonzistencie dát je protikladom tradičných (ACID) transakcií, kde po každej operácií musí byť konzistencia dát zachovaná. Autor v [19] navrhuje použiť perzistentnú frontu správ a rozdeliť databázové tabuľky do skupín podľa ich funkcie (tabuľky pre používateľov, transakcie, správy, atď.). Správy sú z fronty postupne vyberané a operácie opísané správou sú vykonávané. Ak je operácia úspešná, správa sa z fronty vymaže, inak sa opakuje znova. Dočasné nekonzistencie dát umožňuje aj protokol navrhnutý spoločnosťou Amazon, ktorý sa nazýva S3 (z angl. Simle Storage Service) [3].

Avšak, množstvo koncových systémov spravovaných systémom na správu podnikových identít nepodporuje transakcie. Štandardné transakcie, dvojfázový odovzdávací protokol alebo iné tradičné mechanizmy na podporu konzistencie preto nestačia. Navyše, niektoré operácie sú (v systémoch na riadenie podnikových identít) vykonávané dlho, niekedy hodiny ba až dni, z čoho vyplýva, že nie je možné využiť blokujúce mechanizmy. Prístup Saga nie je vhodné použiť, pretože nevieme napísať kompenzáciu ku každej operácií. Tento problém rozoberá aj autor v [9]. Ako možné riešenie prichádza do úvahy využiť model eventuálnej konzistencie so zárukou, že dáta budú v konečnom stave konzistentné. Tento model je využitý pri navrhovaní mechanizmu na podporu konzistencie.

## I.2 Mechanizmus na podporu transakcií v netransakčných systémoch

Cieľom práce bolo navrhnúť vhodný spôsob na vyriešenie nekonzistencie údajov v systémoch na správu podnikových identít. Systémy na správu podnikových identít sa

musia vedieť zotaviť z neočakávaných chýb a nelimitovať pri tom používateľov. Nie je vhodné, aby systémy na správu podnikových identít boli v nekonzistentnom stave dlhú dobu, pretože by to mohlo spôsobiť nesprávne fungovanie systému, ale takisto aj narušenie bezpečnostnej politiky.
Systémy sa správu podnikových identít automatizujú procesy súvisiace s používateľom a jeho životným cyklom v spoločnostiach, od prijatia nového zamestnanca, cez zmenu pozície až po prepustenie zamestnanca. Spravidla, každý zamestnanec musí mať množstvo účtov v rôznych systémoch, ktoré mu umožňujú vykonávať prácu súvisiacu s jeho postavením. Preto aj systémy na správu podnikových identít potrebujú komunikovať s rôznymi koncovými systémami. Tieto koncové systémy potom obsahujú informácie o zamestnancoch a ich prístupových právach. Jeden zamestnanec môže mať účty v rôznych koncových systémoch alebo viacero účtov na jednom systéme.
Účty môžu byť vytvárané viacerými spôsobmi, napríklad použitím centrálneho systému na správu podnikových identít, synchronizáciou zmien z externého systému alebo priradením roly používateľovi, ktorá definuje, ktoré účty majú byť vytvorené. Počas práce s účtami môžu nastávať neočakávané situácie, ktoré vyúsťujú do vzniku nekonzistencii. Podľa spôsobu vzniku nekonzistencii boli identifikované nasledujúce skupiny problémov:
Chyby na koncových systémoch – táto skupina opisuje problémy nastávajúce na koncových systémoch počas prenášania zmien od koncového používateľa, ktoré zadal cez používateľské rozhranie systému. Môže to byť napríklad pridávanie účtu na koncový systém, ktorý je momentálne nedostupný.
Chyby pri synchronizácií – táto skupina opisuje chyby, ktoré nastávajú pri synchronizácií zmien z koncového systému do systému na správu podnikových identít. Napríklad, zmena na koncovom systéme nebola zachytená a tak nebola ani spracovaná.
Závislosti – táto skupina opisuje nekonzistencie, ktoré vznikajú pri práci z navzájom závislými účtami, napríklad účet v aplikácií je závislý na účte v operačnom systéme.
Skupiny – táto skupina opisuje chyby, ktoré nastávajú pri práci so skupinami. Napríklad v LDAP-e je vytvorenie účtu a priradenie ho do skupiny implementované pomocou dvoch operácií.
Práca s rolami – táto skupina opisuje nekonzistencie, ktoré môžu nastať pri práci s rolami. Napríklad máme definovanú rolu, podľa ktorej majú byť vytvorené štyri rôzne účty. Avšak, len dva z nich sa vytvoria a ďalšie dva nie.

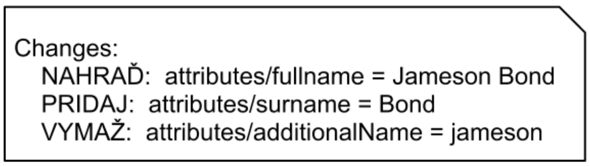
### I.2.1 Navrhovaný mechanizmus

Našou snahou bolo navrhnúť mechanizmus umožňujúci vykonávať operácie na netransakčných systémoch s využitím niektorých výhod tradičných transakcií. Napríklad zavedenie určitého stupňa garancie konzistencií dát (s ohľadom na CAP teorému). Rozhodli sme sa použiť podobný mechanizmus ako pri transakčných systémoch, ktoré sú schopné zotavovať sa z chýb. Navrhovaný mechanizmus sa snaží vzniknuté chyby vyriešiť

kompenzáciami. Ak sa nepodarí chybu pomocou kompenzácie odstrániť, dáta sú vrátené do pôvodného stavu, teda do stavu pred danou operáciou.
Navrhovaný mechanizmus vychádza z modelu eventuálnej konzistencie. To znamená, že systém dovolí dočasné nekonzistencie v dátach so zárukou, že v konečnom dôsledku budú dáta konzistentné. Mechanizmus by však mal byť schopný odhaliť tieto nekonzistencie a pokúsiť sa ich vyriešiť. Mechanizmus navrhnutý v práci je založený na troch základných konceptoch:

- CAP teoréma,
- model relatívnych zmien,
- kompenzácie pre neúspešné operácie.

CAP teoréma hovorí, že nie je zároveň možné garantovať konzistenciu, dostupnosť a odolnosť voči výpadkom uzlov, ale môžu byť vybrané len dve garancie. V našom prípade sme sa rozhodli pre dostupnosť na úkor oslabenia konzistencie. Dôvod nášho výberu súvisí s povahou systémov na správu podnikových identít.
Od systémov na správu podnikových identít je vyžadované, aby boli vysoko dostupné. To zaručí, že je možné stále čítať a zapisovať z/do systému. Navyše je potrebné, aby každá požiadavka mala vhodnú odpoveď aj napriek výpadku niektorého uzla. Nezáleží na tom, či bola operácia úspešná, ale v každom prípade musí byť ukončená a výsledok z nej vrátený používateľovi. Systém musí byť schopný pokračovať v činnosti aj v prípade, ak sa niektorá z odosielaných správ nedostane ku koncovému uzlu.
Ďalším dôležitým konceptom, z ktorého vychádza navrhovaný mechanizmus, je model relatívnych zmien. Model relatívnych zmien sa používa na opis zmenených hodnôt objektu. Namiesto posielania celého objektu pri zmene len jedeného z jeho atribútov, posielame len skutočné zmeny. Tieto zmeny sú vypočítavané na základe hodnôt pôvodného objektu a môžu opisovať tri rôzne typy zmien, ktoré sú zobrazené na obrázku 27.

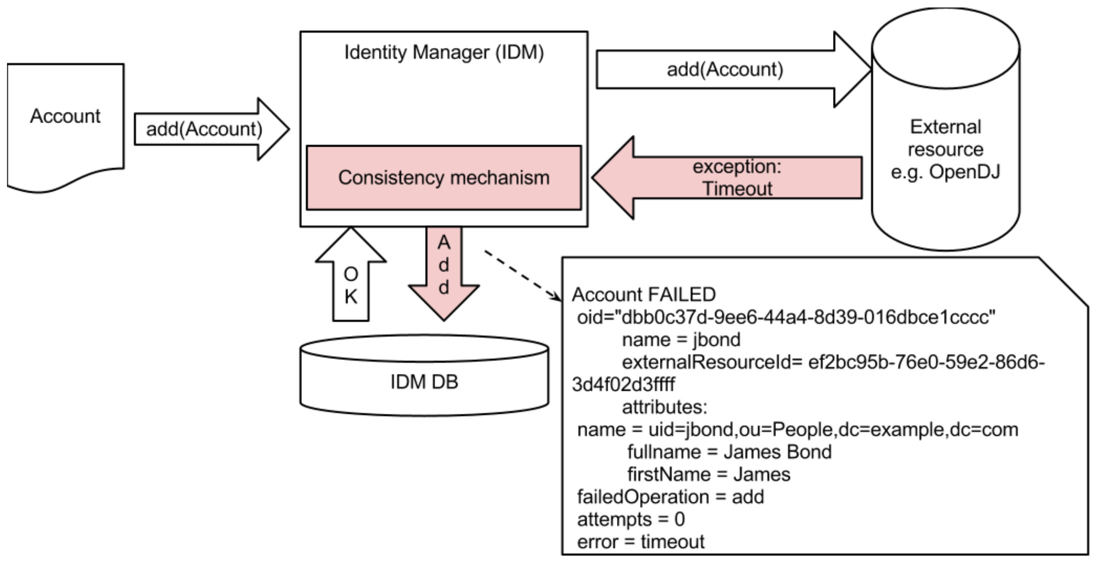


```
Changes:
   NAHRAĎ:  attributes/fullname = Jameson Bond
   PRIDAJ:  attributes/surname = Bond
   VYMAŽ:  attributes/additionalName = jameson
```

*Figure 25 Štruktúra relatívnych zmien.*

Na obrázku XX môžeme vidieť zmenu troch atribútov. Prvý parameter zmien opisuje typ zmeny, a teda, či ma byť atribút pridaný, vymazaný alebo nahradený novou hodnotou. Druhý parameter opisuje, ktorý atribút má byť zmenený a posledný parameter je hodnota atribútu, ktorá je buď pridaná, nahradená alebo vymazaná. Tieto zmeny sú potom aplikované na pôvodný objekt.
Výhoda použitia modelu relatívnych zmien spočíva v tom, že nepotrebujeme uzamykať dáta používané bežiacim procesom. Pri posielaní celého objektu by mohla nastať situácia,

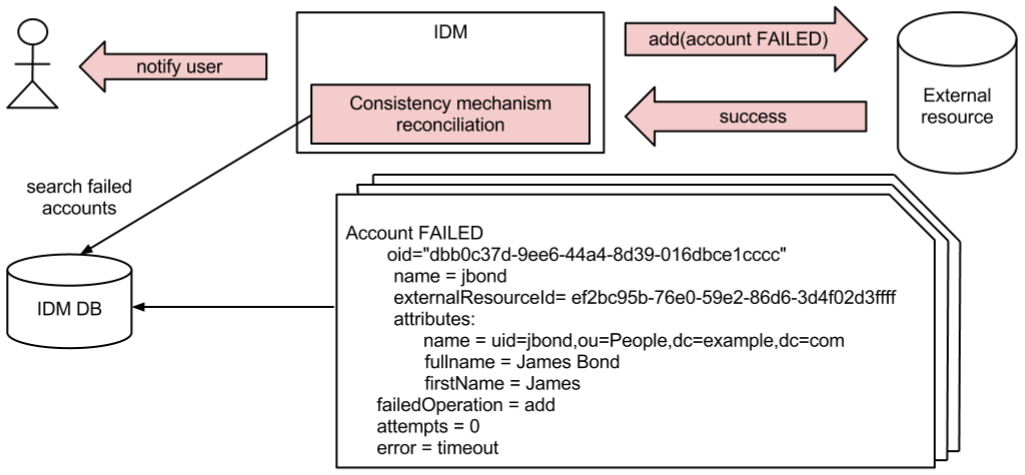
kedy je objekt modifikovaný dvoma rôznymi procesmi skoro v rovnakom čase. Zmenený objekt a jeho nové hodnoty atribútov vychádzajúce z prvého procesu nemusia byť známe druhému procesu. Po aplikovaní zmien druhého procesu by sme tak mohli stratiť zmeny vykonané prvým procesom. Pri modeli relatívnych zmien sú posielané iba skutočné zmeny, to znamená, že ostatné hodnoty zostávajú nezmenené. Ak by sme teda mali dva procesy vykonávajúce sa skoro v rovnakom čase, nestratili by sme tak zmeny ani jedného z nich.
Posledným dôležitým konceptom sú kompenzácie. Kompenzácie sú reakcie na chyby, ktoré mohli nastať. Ich úlohou je snaha eliminovať vzniknutú chybu alebo reagovať na ňu a nájsť vhodný spôsob, ktorý by nenarušil konzistenciu dát. Každá operácia môže skončiť úspešne alebo neúspešne. Ak počas vykonávania operácie nastane chyba, musí sa najprv rozhodnúť či je táto chyba spracovateľná alebo nie. Iba k spracovateľných chybám vieme definovať kompenzáciu, ktorá je v prípade výskytu chyby spustená. Pri nespracovateľných chybách oznámime používateľovi problém, ktorý nastal.
Mechanizmus je navrhnutý tak, aby minimalizoval vznik nekonzistencií v dátach a ak napriek tomu nastanú, mal by na nich vedieť reagovať tak, aby v konečnom dôsledku boli dáta v systéme konzistentné. Mechanizmus bol navrhnutý s ohľadom na jednu z identifikovaných skupín problémov, a to problémy na koncových systémoch. Mechanizmus pozostáva z dvoch častí. V prvej časti sa snaží ošetrovať neočakávané chyby a v druhej časti, nazývanej rekonciliácia, porovnáva objekty uložené v lokálnej databáze s objektami na koncových systémoch. Pri zistených rozdieloch sa ich snaží eliminovať.
V prvej fáze je potrebné vedieť, či je chyba spracovateľná alebo nie. Ak je chyba spracovateľná, má definovaný kompenzačný mechanizmus, ktorý sa v prípade chyby spustí aby vykonal príslušné kroky. Jednotlivé kompenzácie k chybám boli v práci identifikované v sekcii 3 . Ak chyba nie je spracovateľná, znamená to, že nevieme napísať kompenzáciu, ktorá by bola schopná chybu eliminovať. Tieto chyby môžeme považovať za fatálne, a teda oznámime používateľovi chybou, ktorá nastala a ďalšie rozhodnutie je ponechané na ňom. Príklad spustenia kompenzácie je naznačený na obrázku 28.



```
Account FAILED
 oid="dbb0c37d-9ee6-44a4-8d39-016dbce1cccc"
         name = jbond
         externalResourceId= ef2bc95b-76e0-59e2-86d6-
3d4f02d3ffff
         attributes:
 name = uid=jbond,ou=People,dc=example,dc=com
          fullname = James Bond
          firstName = James
 failedOperation = add
 attempts = 0
 error = timeout
```

*Figure 26 Príklad spustenia kompenzácia (kompenzácie, kedy je chyba ponechaná na neskôr).*

Kompenzácia môže eliminovať chybu buď hneď ako vznikne, alebo ju môže odložiť na neskôr. Ak je chyba eliminovaná hneď, mal by byť zachovaný konzistentný stav dát v systéme. V prípade chýb, ktoré nemôžu byť eliminované hneď môže dôjsť k narušeniu konzistencie. Tieto chyby musia byť uložené, aby sme sa k nim vedeli neskôr vrátiť. Na automatické vyhľadanie predtým uložených chýb sme sa rozhodli použiť rekonciliáciu.
Rekonciliáciu môžeme opísať ako proces, pri ktorom zistíme nekonzistencie medzi jednotlivými úložiskami dát. Rekonciliácia slúžia aj na odhalenie predtým uložených chýb a na opätovné vyvolanie nedokončených operácií. Je vykonávaná v pravidelnom intervale (podľa nastavenia) obmedzený počet krát. Ak sa dovŕši maximálny počet behov a operácie sa nepodarí vykonať úspešne, dáta sú vrátené do stavu, v akom boli pred neúspešnou operáciou. Proces rekonciliácie je zobrazený na obrázku 29.



```
Account FAILED
      oid="dbb0c37d-9ee6-44a4-8d39-016dbce1cccc"
       name = jbond
       externalResourceId= ef2bc95b-76e0-59e2-86d6-3d4f02d3ffff
       attributes:
            name = uid=jbond,ou=People,dc=example,dc=com
            fullname = James Bond
            firstName = James
     failedOperation = add
     attempts = 0
     error = timeout
```

*Figure 27 Proces rekonciliácie.*

Netvrdíme, že navrhnutý mechanizmus je dokonalý. Tento mechanizmus bol navrhnutý s ohľadom len na jednu z identifikovaných skupín problémov. Je možné, že pri uvažovaní ďalších skupín problémov bude potrebné mechanizmus rozšíriť o ďalšie kroky. Napríklad, ak by sme uvažovali problémy súvisiace so synchronizáciou, notifikačný mechanizmus by mohol byť užitočný.
Notifikačný mechanizmus by bol vhodný aj pre iné skupiny problémov. Predstavme si napríklad kompenzáciu pre chybu, ktorú chceme nechať na neskôr. Ak sa nepodarí chybu eliminovať po určenom počte krát, dáta ovplyvnené touto chybou sú vrátené do pôvodného stavu. Chyba mohla nastať pri pridávaní účtu a teda výsledkom je, že je účet odstránený z lokálnej databázy. Používateľ ale nemá odkiaľ vedieť, že sa tento účet nakoniec nepodarilo pridať. Pomocou notifikácie by sa to dozvedel, a tak by sa mohol pokúsiť pridať účet znova.

## I.3 Overenie navrhnutého mechanizmu

Navrhnutý mechanizmus bol implementovaný a testovaný. Mechanizmus sme implementovali do existujúceho systému s názvom midPoint. Dôvod výberu práve midPointu bol v práci podrobne opísaný. Implementácia zahŕňa všetky navrhované časti uvedené v práci. Je súčasťou súčasnej vyvíjanej verzie a bude zahrnutá aj do nasledujúceho vydania produktu (midPoint v2.0).

Mechanizmus bol testovaný dvojakým spôsobom, a to manuálne a automaticky. Pri manuálnom testovaní bol midPoint nasadený na aplikačný server Apache Tomcat a testovanie prebiehalo s externým systémom OpenDJ a XML databázou BaseX. Testovanie bolo vykonávané v prehliadači Chrome simulovaním jednotlivých situácií. Konkrétny spôsob testovania jednotlivých situácií bol v práci opísaný v kapitole 4.3.1 . Automatické testovanie predstavujú testy implementované využitím testovacieho prostredia TestNG, vloženej (z angl. embedded) inštancie servera OpenDJ a vloženej inštancie BaseX. Testy sú písané spôsobom „end-to-end“ a simulujú jednotlivé situácie. Pri testoch neboli použité pomocné (z angl. mock) objekty, ale pracovalo sa s reálnym systémom pričom jednotlivé situácie boli vykonávané cez všetky vrstvy systému.

## I.4 Zhrnutie

Práca sa zaoberala navrhnutím mechanizmu na podporu konzistencie údajov pri správe podnikových identít. Prvá časť práce sa zaoberá uvedením čitateľa do problematiky systémov na správu podnikových identít. Sú v nej opísané základné princípy a technológie používané v systémoch na správu podnikových identít. Ďalej sa v nej nachádza analýza existujúcich riešení a takisto analýza existujúcich mechanizmov na podporu konzistencie dát.

V ďalšej časti sa práca zaoberala výberom vhodnej množiny problémov, na základe ktorej by bolo možné navrhnúť mechanizmus na podporu konzistencie dát. Vybraná skupina problémov bola podrobne analyzovaná a opísaná. Na základe nej bol navrhnutý mechanizmus. Ten bol neskôr implementovaný a testovaný. Práca na mechanizme môže pokračovať rozšírením o ďalšiu skupinu problémov.