Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit Nr. 87

Caching Concept for Mobile Engineering Apps

Michael Steffl

Course of Study: Softwaretechnik

Examiner: Prof. Dr.-Ing. habil. Bernhard Mitschang

Supervisor: Dipl.-Inf. Tim Waizenegger,
Dipl.-Inf. Eva Hoos

Commenced: October 1, 2015

Completed: April 1, 2016

CR-Classification: C.2.4, D.4.4, D.4.8

Abstract

Mobile apps in the engineering domain, have to deal with data coming from Product Data Management (PDM)-Systems. This data contains details about the products that are very large. Geometry data like 2D or 3D Computer Aided Design (CAD) representations are included. To get the data, the apps use wireless mediums like WiFi or mobile data networks (LTE, 3G, etc.). Transferring large size data over these mediums take a lot of time and can be aborted through intermittent connectivity. Also the energy consumption increases through the long-lasting transfers. In this master thesis a concept is created that overcomes these problems. A cache on the client is used that stores the relevant data for a fast access. As the disk space on mobile devices is limited, the data that is cached has to be chosen well. Only the data that is currently needed should be stored in the cache and provided to the app. To reduce the waiting times these data should be there before it is explicitly requested. To make this possible the concept of this thesis provides preemptive caching (hoarding). Thereby, the data is cached that will probably be needed next. To decide what data is needed, context is used. The information coming from the environment of the client is used, to derive situations. With the help of these situations the data is determined that gets cached. Besides this context-aware strategy, a traditional way of caching where all requested data gets cached is used in the concept. Furthermore, this thesis addresses the caching mechanism in its entirety. It determines a policy for the replacement of not needed data to free space. Also a strategy for invalidating obsolete data in the cache is determined. Finally, a prototypical implementation of the concept within an existing mobile engineering app is presented. With the help of this prototype the concept is evaluated.

Contents

1 Introduction 9

2 Basic Concepts 11
- 2.1 Product Data Management 11
- 2.2 Caching 13
- 2.3 Mobile Apps in the Engineering Domain 20

3 Related Work 27

4 Caching Concept 29
- 4.1 Cache Granularity 29
- 4.2 Cache Selection Strategy 31
- 4.3 Cache Replacement Policy 36
- 4.4 Cache Coherence Strategy 40
- 4.5 Conclusion 42

5 Prototype 45
- 5.1 The Existing App 45
- 5.2 Overall Architecture 48
- 5.3 Implementation of the Caching Concept 56

6 Evaluation 71

7 Conclusion and future prospects 79

Bibliography 81

List of Figures

2.1 Example-Product structure 12
2.2 Virtual Prototype [WPG14] 13
2.3 Overview of caching mechanism 16
2.4 Types of cache granularity 17
2.5 Simple architecture for mobile apps in the engineering domain 21
2.6 Logical data model 22
2.7 Mapping from logical to physical data model 23
2.8 Basic Context Model for the Engineering Domain [HSM16] 25

4.1 Used cache granularity 30
4.2 Context-aware data provisioning 32
4.3 Context-aware selection - location based selection [HSM16] 33
4.4 Dynamic Context-Aware Caching - most popular 35
4.5 Gain function based replacement strategy 39
4.6 Stateful Invalidation Protocol (SIP) 41
4.7 Overview of the caching concept 43

5.1 Architecture of the existing app 46
5.2 Data model of the protoype 47
5.3 Architecture of the prototype 49
5.4 Cache of the prototype 52
5.5 EER model of the database 54
5.6 Process for showing product components 58
5.7 Protocol - getResource 61
5.8 Protocol - getResourceSet 63
5.9 Replacement policy 65
5.10 Initialization at the server 67
5.11 Protocol - invalidation 68

6.1 Use case for the evaluation 72

List of Tables

4.1 Prioritization of the resources in the cache 37

5.1 Priority list 56

6.1 Measurement of scenario 3 74
6.2 Measurement of scenario 2 75
6.3 Measurement of scenario 3 76

List of Listings

5.1 Manifest file for caching the core data of the app 52

1 Introduction

Mobile computing and the use of apps running on mobile devices is getting more and more popular in the engineering domain [HGM15]. In most of the cases the apps need access to the data of PDM (Product Data Management)-Systems. In these systems all relevant product and process information across the product life cycle are stored [HSM16]. These information consist of meta data describing the products and also geometry data, like 2D and 3D CAD (Computer Aided Design) representations that can be very large. This is especially true in the automotive sector in which models consist of hundreds of components.

Dealing with this kind of data in apps running on mobile devices is a challenging task [HSM16]. Accessing product data of PDM-Systems can lead to long loading times due to the data size and the low bandwidth of the wireless communication. This also becomes noticeable in the consumption of energy that is limited on mobile devices. The transmission of data is expensive. Through intermittent connectivity caused by interferences or by the lack of wireless coverage, it can happen that the transmission of the data is aborted. During this offline time, it is not possible to work with the mobile app. These technology-oriented problems makes it difficult to work with mobile apps in the engineering domain [Rot15] [PS12].

To overcome these problems, an implementation of a cache on the client is used. A cache is a component where data is stored that can be retrieved from there later upon requests [BBFS11]. In this master thesis, a concept for the use of a cache within mobile apps in the engineering domain is created. The concept describes a caching mechanism that is used to reduce the waiting time within the app. This is realized through storing the needed data in the local cache of the client. Accessing data from here, is much faster than transferring them over the network. Already cached data has not to be requested from the server again. This also reduces the amount of network transmissions that affect the consumption of energy. The device spent less time in receiving data over the network. This saves energy. Through the caching of data, the problems caused by intermittent connectivity are also reduced. Cached data can be used, even if the connection to the server is not available. To make this possible, the concept provides some techniques and strategies. Besides caching all requested data,

these allow selecting and caching data, before it is explicitly requested. This is called preemptive caching or hoarding [HSM16].

To decide, what data will be hoarded, context is used. Context is the information that is used to characterize a situation of an entity [ADB+99]. In the engineering domain this information come from the users (entity) that interact with the mobile app within the domain. This information is used to derive the situations that helps to determine the data that is currently needed. It is provided in the cache, before it is requested by the users. This represents context-aware caching. The cached data of the app adapts to the current situation.

The developed concept is designed for a concrete mobile app that is used in the engineering domain. The app is executed in a web-browser (web app) and allows to view 3D models of products. However, the concept can be used for any other app in the engineering domain. Summarized the caching mechanism of the concept provides the following key features:

- Reduced waiting time by use of caching.
- Fewer network transmissions by use of caching.
- Robustness against intermittent connectivity by use of preemptive caching.
- Usage of hoarding to enable an offline mode.

Structure of the thesis

The remainder of this master thesis is structured as follows: Chapter 2 provides basic knowledge about PDM-Systems, the engineering domain and caching in the large. In Chapter 3 concepts of other works are explained that address with providing data for apps on mobile devices. The caching concept that represents the core of this thesis is described in Chapter 4. The realization of the key features of the caching mechanism are explained. After this, the implementation of the concept is presented in Chapter 5. This is done within a prototype that is used in the engineering domain. This is evaluated in Chapter 6, before the thesis is completed through the conclusion and future prospects in Chapter 7.

2 Basic Concepts

In this chapter the basic concepts for the thesis are described. Before going into details of caching that is used within the data management, the PDM-Systems are explained. Mobile apps in the engineering domain interacts with such systems. Therefore, it is essential to understand how they work and how the data look like. After that, basic information about caching is provided. An overview over the types, the techniques and the characteristics of a caching mechanism is given. Finally the mobile apps in the engineering domain are explained. The architecture, the data model and the context of the engineering domain are considered.

2.1 Product Data Management

The Data that is needed on mobile apps in the engineering domain are basically stored in *Product Data Management (PDM)*-Systems. They are used to manage the product data across the entire product life cycle. A combination of the product specific data and the corresponding process management allows a complete reconstruction of the product in arbitrary construction states. The base of the PDM are established by the product structure which arises during the product life cycle [SI08]. All components of the product and the relation to other components can be found here. With this structure it is possible to get an detailed overview over the whole product. An example for such a product structure is presented in Figure 2.1. This shows a trimmed-down version of a motorboat in a tree shape that consists of components and sub-components.

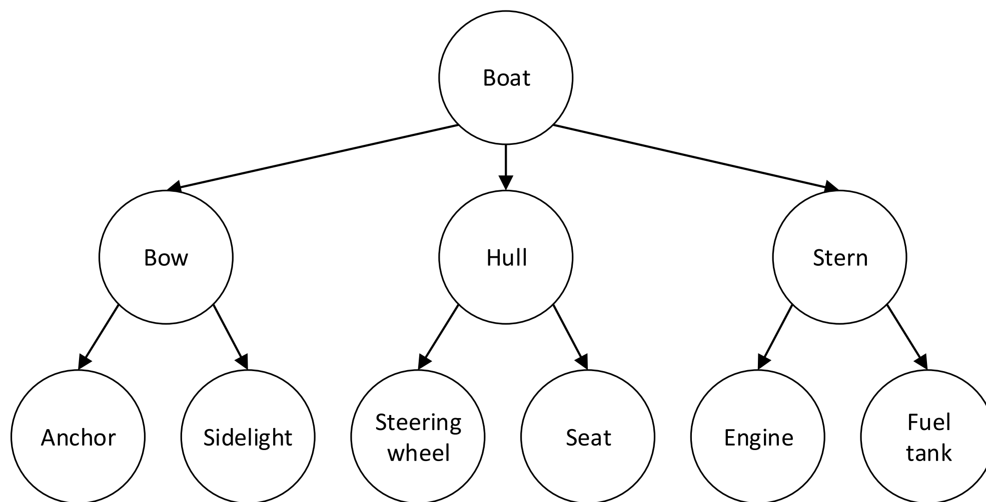

Figure 2.1: Example-Product structure

The Figure shows three levels of the tree. The first level is composed by the root node *Boat*. This node has three child nodes *Bow, Hull, Stern* that represents the sub-components of *Boat*. All of these have child nodes, too. This goes further until all nodes of the product can be found in the tree.

This product structure composes one of the base modules of the *virtual prototype* [WPG14]. A *virtual prototype* is a virtual representation of the whole product. It is possible to simulate the whole behavior of the product. On that way it can be used to do a lot of different analysis and tests virtually. This can save a lot of money and time, as the first tests and analysis can be done before manufacturing a real touchable prototype. Thereby the *Virtual Prototype* consists not only of the product structure. Figure 2.2 presents all the parts of the it and shows the correlation between them.

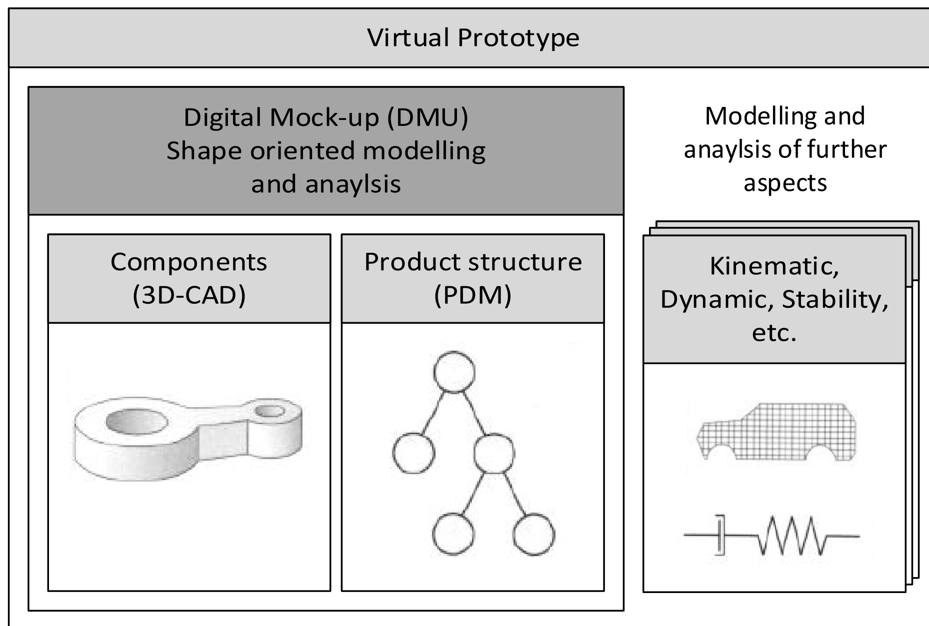

Figure 2.2: Virtual Prototype [WPG14]

The main part of the *Virtual Prototype* is presented by the *Digital Mock-up (DMU)*. A DMU is an aggregation of single components to a 3D model of a product [WPG14]. With the help of the *product structure* and the position and orientation information (*3D-CAD*) of the *components* this aggregation takes place. With this *DMU* it is also possible to do some like assembling, kinesic behavior or stability of the product. Through the integration of further aspects (thermal, shape, etc.) the *DMU* becomes a *virtual prototype* [WPG14].

2.2 Caching

In the following Section basic information about caching is given. It is described what types of caches exists and when they are used. After that, two different kinds of caching techniques are explained. It is described how they differ. For the realization of a cache within an app, a caching mechanism is needed. This is composed out of different characterizations. The details about this, are explained after the caching techniques. Thereby, an overview of possible strategies that can be used within the characteristics is given.

2.2.1 Types of Caches

A cache is a component that stores data that can be retrieved from there later upon requests [BBFS11]. The aim is to minimize the access times and to reduce the amount of accesses to a slower background medium. To achieve that a good cache hit ratio is aspired. That means that as much as possible requests are responded directly from the cache. If a request is responded from the cache, a cache hit is achieved. The hit ratio is computed out of the number of requests divided by the amount of cache hits during the requests [FZJF06] [FCAB98].

There are many ways to realize a cache. This depends strongly on the purpose of it. It is distinguished between caches that accelerate hardware and caches that accelerate software. In this thesis only the caches with regard to software are considered. The three types that are considered are *server caches*, *network caches* and *client caches*.

Server caches are placed in front of a server with the aim to catch the requests and try to serve the data directly from the cache. When a client requests data that has to be computed on the server, it gets cached. The next time the same data is requested again, a new computation is not necessary. The data can be delivered directly from the cache. This reduces the server load.

Network cache is the second type. The cache is located at a position in the network. The aim of this approach is to store common requested data in a cache that is near to the clients. This enables a faster access to the data. The most popular representatives from this kind of caching are proxy servers and content delivery networks (CDN). Proxy server store often accessed data within the network and try to deliver all the data that is requested directly from its store. This will improve the response times, as the proxy server is located nearer to the clients than a server from a probably far away server. Content delivery networks use distributed networks of proxy servers with the aim to additionally reduce the response times in huge networks like the internet.

When there is a need for an individual cache, a *client cache* can be useful. Individual cache means that every client has it's own cache. Only the data that is needed from the client is stored there. The data is cached directly on the client on the local disk. Software applications can access the data without requesting it from a server when they are stored in the cache. Modern browser like Google Chrome, Internet Explorer or Firefox uses this kind of caching to speed up loading pages from the internet. The first time a user requests a page, all the needed data from it will be stored in the browser cache. When requesting the same data again, they can be loaded from the cache on the storage.

2.2.2 Caching Techniques

Caching data for mobile apps can be realized differently. There exist several techniques for doing that. In this thesis it is distinguished between *traditional caching* and *preemptive caching (hoarding)*. The key difference between those two techniques is the selection of the data that should be cached. In traditional caching the data is only cached, when it is explicitly requested. E.g. this can occur through a user interaction like clicking on a load button that starts loading an image from a server. When the image arrive at the client, it gets cached and can later be accessed directly from the cache.

The other technique of caching is to store the data preemptive. Preemptive caching means that there is no need for an explicit initial request. The data gets cached, before it is requested [HWM09]. On this way, the caching can be improved, as the data can be accessed directly from the local cache, even for the first time. This kind of caching is also known under the term hoarding as all the data is hoarded in the cache for later usage [KR01]. However the challenge with this kind of caching is to identify the data that will probably be needed in future. This selection of the suitable data is very important in this approach.

2.2.3 Caching Mechanism

To realize a cache within a mobile app, there is the need to define some rules and strategies. E.g. it is necessary to define, what kind of data should be cached, and what to do when the cache is full. These rules and strategies combined constitute the caching mechanism. Rathore and Prinja [RP07] create a characterization where these things are treated. They define characteristics that make up the caching mechanism. These are cache granularity, cache coherence strategy and cache replacement policy. In this characterization there is missing a single characteristic that gives the information which data is selected to be cached. This is important to distinguish between the two caching techniques *traditional caching* and *preemptive caching (hoarding)*. The main difference between them are the selection of the data that should be cached. Therefore the cache selection strategy is added. An overview of the characterization of the caching mechanism is presented in figure 2.3. This overview includes the different strategies of the single characteristics. In the following section, the details about the characterizations and their strategies are described.

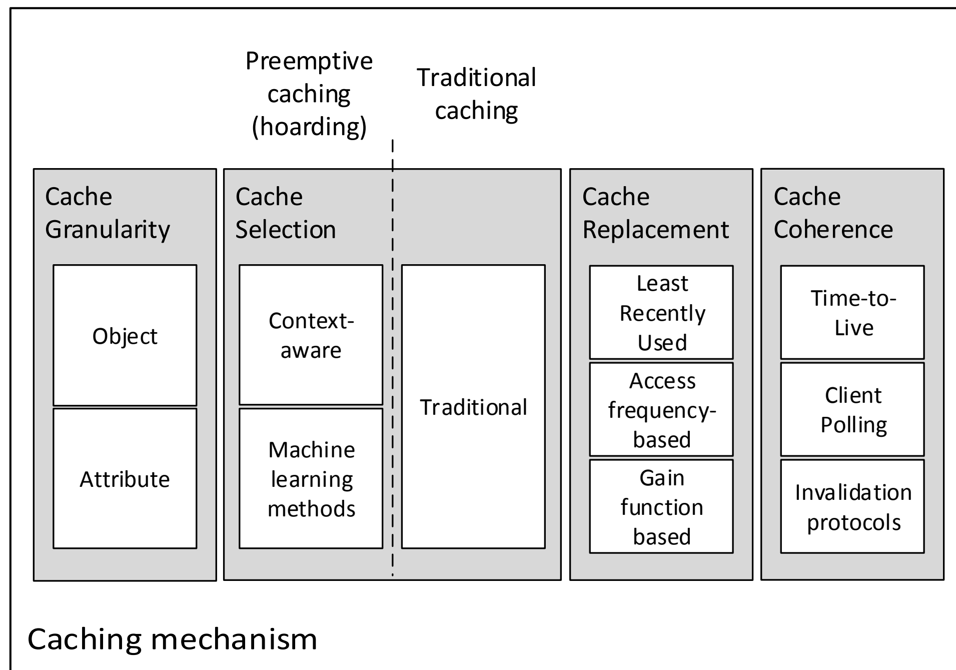

Figure 2.3: Overview of caching mechanism

Cache Granularity

The cache granularity composes the base of a caching mechanism. It determines, what kind of data is stored in one cache entity. A cache entity is the smallest unit in the cache. It is the physical form of the data that is cached [Tan07]. Choosing the best fitting granularity can make the caching mechanism more efficiency. A granularity can be chosen in this way that the whole information stored in one entity is needed. In this case, the cache space is used in an optimal way. If too many information within a cache entity is not needed, space is wasted. That space could be used from other data [RP07].

The determination of a granularity depends strongly on the used data model. There is no general way of determine a granularity. An abstract transferable way that fits to the most data models, is the use of objects and attributes. A high level view is presented in figure 2.4. The differences between object and attribute granularity is shown.

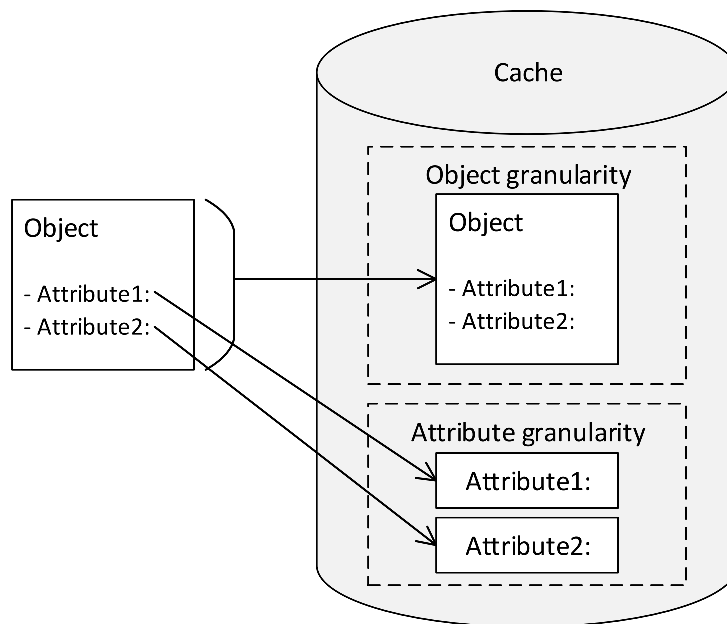

Figure 2.4: Types of cache granularity

An object consists of multiple attributes that contain different information. When using the *object granularity*, the object with all its attributes is stored in on one entity. The object gets the smallest unit in the cache. This can be helpful in scenarios where most of the attributes from one object are needed in conjunction. When clients are only interested in some attributes of the objects, a better way is to use the *attribute granularity*. In this granularity an attribute is the smallest unit in the cache. This can lead to a better utilization of the cache space [RP07].

Cache Selection Strategy

The cache selection strategies determines, which data have to be stored in the cache. As the space on mobile devices is limited, in most of the cases it is not possible to store all available data in the cache. Only parts of it can be stored. It is important to select the data that is relevant and that leads to a good cache hit ratio. For selecting the data the two different techniques can be used. The data can be selected on a traditional way, or through preemptive caching. Several strategies exist for the selection:

- Cache all requested [ADB+99]
- Context-aware selection [FZJF06]
- Machine learning based selection [SSAS11]

Cache all requested represents the way of traditional caching. Within this selection strategy only the data that is explicitly requested by a user is stored in the cache. Once the data is requested, it gets cached and can be accessed directly from the cache the next time it is needed. On this way, the frequent used data will be cached [ADB+99].

Context-aware selection means that the selection of the data that is cached, adapts to the current situation [FZJF06]. This can be used for preemptive caching. Depending on the current situation, data is requested in the background and stored in the cache. It is hoarded for a later usage. When the data needed in the situation is requested by a user, this is still cached. It can be accessed directly from the cache. Which context is considered, depends on the used context model. E.g. it is possible to use the situations that are derived from a location. Thereby, the cached data adapts to the current location. For a smooth running it is necessary that the situations are recognized early enough. If they are recognized too late, it is possible that some data is still not cached.

Another way of preemptive caching is the use of *machine learning methods*. Thereby Classification and Regression Trees (CART) can be used. CART allows to identify patterns and relationships that can be used for preemptive caching [SSAS11]. This can be used like a recommendation for data. The server knows through the access patterns, which data is requested together. When a client requests certain data x, the server checks for data that is requested together with x. When it can find some, the server sends these together with x back to the client.

Cache Replacement Policy

When the limit of the cache is reached and there is no more space left for new incoming data, a policy is needed that determines which data can be replaced. Without such a policy a caching mechanism won't work, as once a cache is full no other data has the chance to get into the cache. Three possible policies are described:

- Least recently used [JJ99]
- Access frequency-based schemes [LS97]
- Gain-based function [XHLL04]

Least recently used considers the access times of the data in the cache. When the cache is full and a new data arrives at the client, the access times of the data in the cache are compared. Depending on the size of the new data, one or multiple data entries that have the oldest access times are selected. The selected entry get deleted and the new data can be stored [JJ99].

Access frequency-based schemes can be used for realize a replacement policy. Instead of considering the access times, different computations on the access frequencies will be done. E.g. the mean over all access can be computed. In return, every data in the cache gets a access counter. When the data is accessed, the counter will be incremented. If the cache is full and a new data should be cached, the mean is computed. The data entry with a lower access frequency than the mean will be replaced. With this policy it can happen that the cache is crowded with old items that have a high counter. In this case it is not possible for new incoming data to stay in the cache after the next replacement. If this behavior is not desirable, this policy have to be adapted. The mean have to be computed only over a certain time window instead of the overall time. On that way old objects that are not used currently do not fall into the weight [LS97].

Gain-based function is also a policy that can be used. For selecting the data that will be replaced, a gain function with different parameters are defined. The parameters can be the access probability, the data retrieval delay, the update frequency or the data size. These can be combined, or can be used separately. For example it is possible to determine that the data over a certain size and over a defined update frequency should be replaced [XHLL04].

Cache Coherence Strategy

When caching data it is necessary to guarantee that it is up-to-date when working with it. For this reason there is a need for a strategy that allows to identify obsolete data and guarantee a consistent state in the cache. Thereby, it is distinguished between weak consistency and strong consistency. The first one means that there is some level of acceptance where inconsistent data is allowed. For example for a certain time. Strong consistency instead do not allow this. The consistent state have to be reached as soon as possible [Aro08]. On mobile devices strong consistency is related to some constraints. Because of intermittent connectivity it can only be guaranteed during the online time of the device. The other way would be to block all data when being offline. To realize consistency three strategies are possible:

- Time-to-Live (TTL) [Aro08]
- Client Polling [Aro08]

- Invalidation protocols [BI94] [JEHA97]

Time-to-Live can be used to guarantee weak consistency. It is realized through a lifetime that is given to every data that is cached. At the point in time the data is stored in the cache, the lifetime begins to run down. As long as there is time left, the data can be accessed from the cache. When it's over, it gets deleted and has to be requested again from the server. Weak consistency is given, as it can happen that during the lifetime of the data an inconsistent state is reached. This occurs through a new version of the data on the server during this lifetime. Consistency is rehabilitated after the time is over and the data has to be requested again [Aro08].

Client Polling guarantees strong consistency. Every time the client accesses data from the cache, a request to the server is initiated. It is checked, if the data is up-to-date or if a new version exists. Thus it can be guaranteed that the data is the newest one. Here the constraints of mobile computing take into effect. A consistent state can only be guaranteed during the online time. Otherwise the data has to be blocked when there is no connection. That would exclude an offline mode [Aro08].

Invalidation protocols use messages to inform the clients about invalidate data. They are sent from the server and contain a report about data that have been updated. Thereby, it is distinguished between *stateful* and *stateless*. In *stateful* approaches the server knows the clients and its cached data. It is possible to send the invalidation report selective to the clients. *Statelss* servers do not know anything about the clients. Therefore the invalidation reports have to be sent to all clients over a broadcast channel [RP07]. The invalidation reports can consist of different information that is used to invalidate the data. Barbara et al. uses timestamps [BI94]. Jing et al. instead bit-sequences [JEHA97].

2.3 Mobile Apps in the Engineering Domain

The cache should be implemented to overcome the technology-oriented challenges of mobile apps in the engineering domain. For designing a caching concept, it is important to know how these kind of apps look like. Especially information about the architecture and the used data model are essential. The architecture is needed to decide what type of cache should be used. The used data model has strong influence on the used cache granularity that forms the root of a caching mechanism. Additionally the context of the engineering domain is considered. With the help of context the different situations of the domain are derived. This can be used to determine the

data that is needed. In the following Section the architecture, the data model and the context of the engineering domain are described.

2.3.1 Architecture

Mobile engineering apps get the data from back-ends that stores the relevant product data. The different clients, have to communicate with a server. In figure 2.5 the high-level architecture of such a system is shown. Thereby, the details are renounced and only the relevant information is presented.

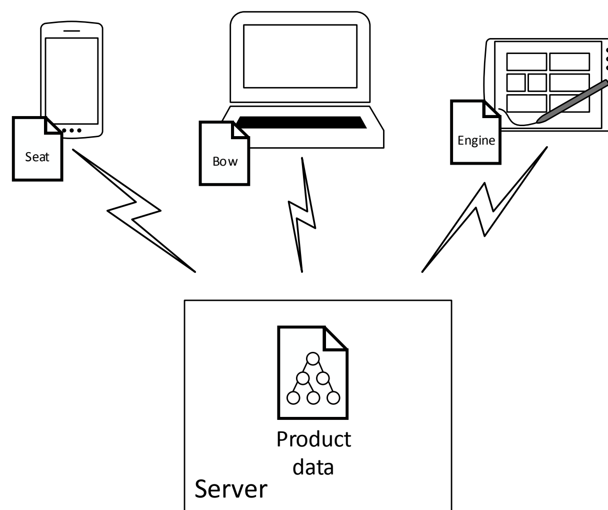

Figure 2.5: Simple architecture for mobile apps in the engineering domain

The apps are designed for running on any kind of device. So it is possible to have a short view on small screens (smart phones), but also have a more detailed view on larger screens (notebooks). The users of the app are interested in product data coming from PDM-Systems. To get the data, the clients requests it from the server over a WiFi connection. So the app can be used anywhere within the company (supposed a WiFi connection exists). Different clients request different components of the product data (*seat, bow, engine*). This depends on the task that is executed with the client or other circumstances that require specific data [HSM16].

2.3.2 Data Model

Mobile engineering apps process product data coming from PDM-Systems [HSM16]. They include information that can be used to reproduce the product. To describe the data model it is distinguished between the logical and the physical data model. The logical model describes the structure of the physical data.

Logical Data Model

The logical data model of the product data is represented by a tree [SI08]. A tree is composed of inner nodes and leaf nodes. Thereby the inner nodes are used to divide the product data in its sub components. They build the branches of the product. The leaf nodes include the information about the components. They always belong to a branch. In figure 2.6 the logical data model is explained. For this an extract of the product data of a boat is used. In return, three levels containing the nodes of the tree are presented.

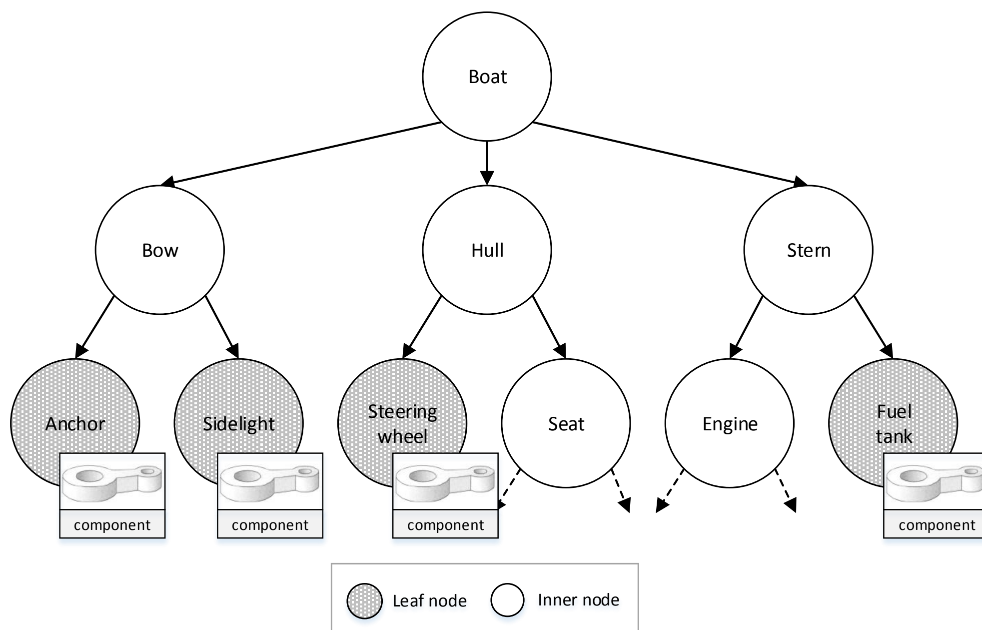

Figure 2.6: Logical data model

The extract starts on the first level with the *inner node boat*. This node represents the root. It is divided into 3 branches that are located on the second level. They are induced by the inner nodes *bow, hull and stern*. On the third level some of the components of the boat are located. In the case of *anchor and sidelight* they belong to the branch that is represented by *bow*. *Hull* consists of a further inner node (*seat*). This starts a new branch which means that on the fourth level broader nodes will follow. With this tree structure it is possible to describe the whole product and its single components. E.g. it can be expressed that the *bow* of the *boat* is composed by *anchor* and *sidelight*.

Physical Data Model

To make the different information of the products available for the apps, they have to be stored physically. For the physical representation of the product data, the term resources is used. It is distinguished between the structure and the single components. Both information has to be provided in a resource. How the logical data model is mapped to the physical is shown in Figure 2.7. This is done by the example of one product.

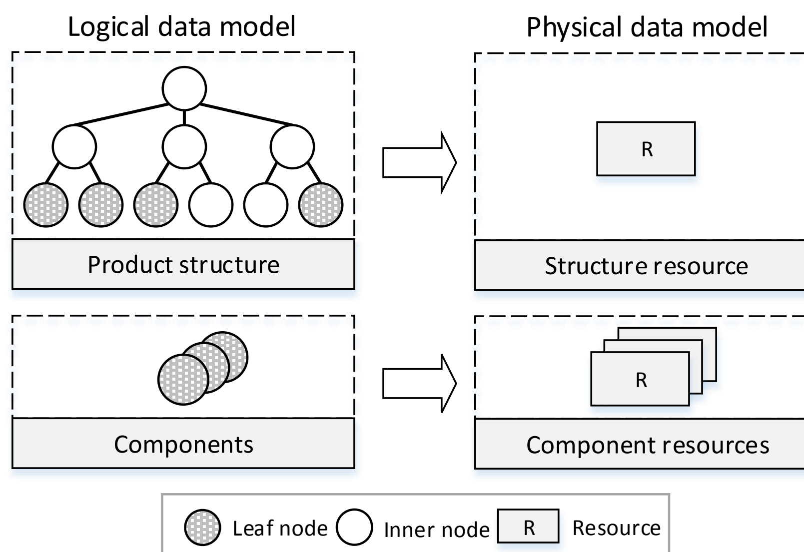

Figure 2.7: Mapping from logical to physical data model

One product is represented by a structure and multiple components. The *nodes* and the relations from the logical data model compose the structure of the product. To make this structure available for the mobile app, it is stored in one single *resource*. With this *structure resource*, it is possible to arrange all the components of a product in the correct order. All references to the single components are included. Additionally to this structure, all the *leaf nodes* are stored in resources. These nodes represent the components of the product. In these *component resources* all the information about the components are stored. These include the meta data and the geometry data (CAD). When talking about the product data, the term resource is used up to now.

2.3.3 Context of the Engineering Domain

Context is any information that can be used to characterize the situation of an entity, which could be a person, a place or an object that is relevant [ADB+99]. When a user within the engineering domain interacts with an app, such information is created. E.g. when the position of a user using the app is tracked, this position information can be used to describe the current location. This location represents a current situation. For identifying the context of the engineering domain, context models can be used. Hoos et al. create such a context model for the engineering domain [HSM16]. This is presented in Figure 2.8. The context model consists of different dimensions that describe the context. To refine the dimensions, context elements are used that are linked together over relations. These elements can get different values that are used to describe the current situation. Therefore the values of the elements are used, to derive situations [HSM16].

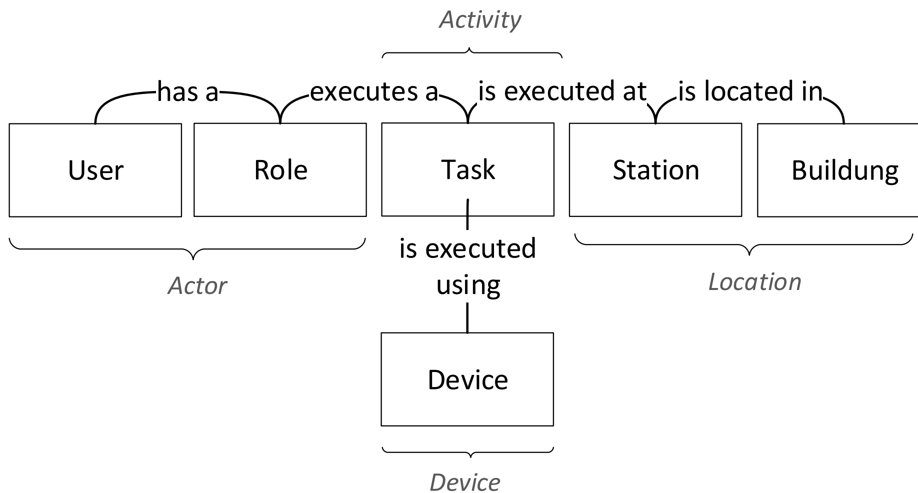

Figure 2.8: Basic Context Model for the Engineering Domain [HSM16]

The context model consists of six context elements that are represented by the rectangles. Each of these elements has a relation to another one. The *task* for instance is related to the *station* over *is executed at*. The elements and the relations are the result of the refinement of dimensions that are identified as important for the engineering domain. That way the dimension *actor* consists of the two elements *user* and *role* that are related through *has a*. Another relation *executes a* creates a link to the dimension *activity*. So things User A that has the Role B executes Task C at station D can be captured. With the help of this context model, it is possible to derive the different situations of the engineering domain. This situations can be used to adapt the app and its resources to the current needs [FZJF06].

3 Related Work

This chapter presents several works that address providing data on mobile devices. It is examined if the approaches of the other works can be used for the caching concept within the mobile engineering app.

In [GAAU15] they create a reliable, consistent and efficient data synchronization for mobile apps with Simba. Simba is a data-sync service which provides a SDK for developer to manage the data within an mobile app. Simba consists of a client (sClient) and a server (sServer). Within the client the local storage is located. The storage consists of a table storage (SQLite) and an object storage (LevelDB) where the documents are stored as key-value pairs. Therefore it is possible to store any kind of data. The sClient acts as a proxy on a mobile device. All apps using the SDK communicates over the sClient with the sServer. The communication between the sClient and the app is realized with local RPC, in Android with AIDL. It also gets all the messages from the server and notifies the app when new data arrive or when a conflict exists. For synchronization and conflict detection flags in the rows of the data tables are used [GAAU15]. As the sClient is currently only implemented on Android it is not suitable as there is a need for a platform independent caching mechanism that can be used within web browsers.

Another approach where the data on the client is stored in a local database which is synchronized with the server, is presented in [PBHS11]. Here they implemented a new client centric protocol for database replication in mobile environments. For the synchronization and communication with the server REST services are used. They are implemented as a web app acting as a middleware between the client and the database located on the server. With the help of last-update timestamps the client requests all the data from the server that were updated since the last synchronization. As it is possible to change data on the client and push it to the server when online (write-any approach), an app specific conflict resolution is implemented, supporting eventual consistency [PBHS11]. This approach uses a replication of data that refers to the whole database. This is not feasible for caching mechanism within the engineering domain. The data is too large and the disk space on the mobile clients too small. A similar approach is used in [SMC+14]. They create an algorithm for mobile replicated database management

synchronization (MRDMS). The difference is, that the synchronization is cell based. That means that every cell in the database have a timestamp. For this purpose every table gets a corresponding timestamp table. To identify updated resources, the cells and the timestamps are transformed in a specific format matrix. At synchronization time JSON over HTTP is used for exchanging the data between the clients and the server. As all the data is replicated, this is also not feasible.

To avoid the replication of the whole database, Gollmick et al. introduced a service using replication views in [Gol03]. Replication views are client defined subsets of server databases the client want to store locally. After the synchronization with the server, the client can work with the data in the subset offline. When the client has updated the data locally, they have to be synchronized with the server when being online again. For this they use the capabilities of the used client and server Database Management System. This service is designed for the use with relational databases. The implementation is based on a three-tier architecture where a replication proxy server (RPS) is placed between the client and the server. The RPS contains an additionally replica with the intention for scalability, security and reliability. As this service is designed for relational database that approach do not fit. The data that is considered within the caching concept contain the product data that includes geometry data (CAD). These data is not appropriate for storing it in relational databases.

Hoepfner et al. create a flexible framework (extension to MyMIDP) for caching data by the means of SQL queries [HWM09]. The requests to the server are realized directly as SQL statements. Before it is sent to the server, the query is splitted after a certain scheme with the goal to get a key that can be used for searching still cached data on the client. When the data can't be found, the query is prepared to comply with the used strategy and is sent to the server. Possible strategies are for example semantic or preemptive caching. When the server delivers the answer in form of a record set to the client, it is stored in the cache and is tracked by a cache handler. The cache handler is responsible for delivering the resources when they can be found by the key. This approach has also the problem that it is deigned for SQL queries.

The use case of the caching concept that is developed for mobile apps in the engineering domain is very particular. A common approach can not be applied to it. Therefore, it is necessary to develop a new concept.

4 Caching Concept

In the following chapter, the developed concept for realizing the caching mechanism within a mobile engineering app is described. The used type of the cache is a *client side cache*. As the mobile apps communicates over wireless connections with the back-end, this is the most feasible. The resources are stored directly on the device and are accessed from here. Once cached, the resources can be used without requesting it from the server.

For the development of the concept, the different characterizations of a caching mechanism are used. In Section 4.1 - *Cache Granularity*, it is explained, what kind of data is stored in the cache. The selection of the resources that have to be cached is described in Section 4.2 - *Cache Selection*. Thereby, context-aware and traditional selection strategies are used. When the limit of the cache space is reached, a policy is needed that decide, if a resource in the cache should be replaced to free space. It is also essential to select the most meaningful resource to replace. The used policy within the concept is explained in Section 4.3 - *Replacement Policy*. To guarantee that the resources in the cache are up-to-date, a strategy is needed that invalidates obsolete resources in the caches of the clients. This strategy is described in Section 4.4 - *Cache Coherence Strategy*. At the end of this chapter, a summary of the used strategies within the mechanism is given in Section 4.5 - *Conclusion*.

4.1 Cache Granularity

For the determination of the cache granularity, the data model from Section 2.3 - *Mobile Apps in the Engineering Domain* is used. The resources that have to be provided by the cache, is the product data. Physically it consists of the both structure and component resources. Each of these resources can be seen as an object with different attributes. A structure resource contains other attributes than a component resource.

When working with the product data within the app, it should be achieved that the overall structure of a product is available. Also the single components of it should be visualized completely with regard to the geometry data. For this reasons every

resource has to be provided by the cache completely. That means that a resource is the smallest unit that is stored in the cache. As a resource can be seen as an object, the granularity of the caching mechanism is object. An abstract view of the cache granularity is shown in Figure 4.1 by the example of the boat. It can be seen, what kind of objects are stored in the local storage of a client.

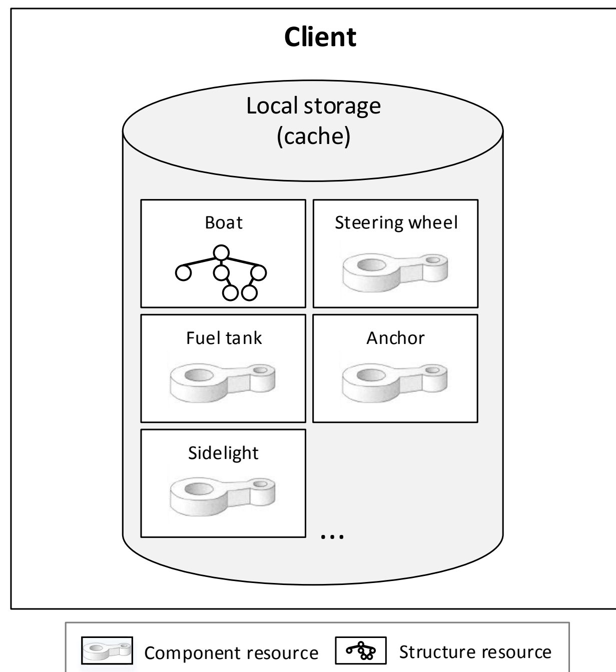

Figure 4.1: Used cache granularity

Each client has its own *cache* that is located in the *local storage*. When there is the need to cache resources, always the whole resource is stored. In this view, the *cache* includes the structure information of the boat that is included in the *structure resource boat*. Additionally to this resource all the components (*fuel tank, sidelight, steering wheal, anchor*) of the boat are stored as separate *component resource* containing the meta data and geometry data.

4.2 Cache Selection Strategy

For the selection of the resources that are cached, preemptive caching in combination with traditional caching is used. Preemptive caching is realized by context-aware selection. With context it is possible to determine the individual needs of a client that is used by a user. The cached resources adapts to the current situations. If the current need for resources are covered in the cache, the cache hit ratio increases. The aim of machine learning methods like CART instead, is to reflect the common needs. It uses history data to cluster resources and give common recommendations. They do not adapt to the current individual needs as fast as the context-aware selection do. For this reason context-aware selection is used in the caching mechanism.

To additionally increase the amount of cache hits, traditional caching is used in parallel. All user requested resources are also stored in the cache. On that way the unused cache space is used to store frequent accessed resources. When the different strategies conflict cause of missing cache space, it is the job of the replacement policy to decide, which resources are kept in the cache. For all the strategies it is prerequisite that during the caching of a resource a connection to the server exist. Without a connection it is not possible to request resources and therefore they can't be cached. After caching it is also possible to use them offline.

In the following Section, context-aware and traditional selection are described. Thereby, different strategies that considers different context elements are used for context-aware selection.

4.2.1 Context-Aware Selection Strategy

Context consist of several elements within the engineering domain. To create a context-aware caching within an app, it is necessary to identify the elements that are used to determine the selection. For this purpose, the context model from Section 2.3.3 is used. As not all dimension of the model are useful for a caching mechanism, only the relevant are used. These are *actor*, *activity* and *location*. Within these dimensions, the context elements *role, task, station and building* are used. Also the relations are not considered. The context elements are considered separately. If multiple situations occur at one moment (more than one context element has a value), then multiple active situations exist. E.g. is is possible that a specific task is executed near a station. Now, both the context elements have a value.

To make it possible that the cached resources adapt to the current situation, the corresponding resources has to be defined. For this purpose, individual resource sets

for the different situations of the context elements are created. In these resource sets, all the resources that are needed in the situation are included. Therefore it is necessary to create links between the resources and the situation. How this is realized, can be seen in figure 4.2. Thereby the process and the involved components of the client and the server are shown.

Figure 4.2: Context-aware data provisioning

On the server side the information about the *links* are stored. These include the information about the situation and the corresponding *resources* that are available on the server. To make them available for the clients, *resource sets* are created out of the links. One *resource set* includes thereby all the resources that are linked to the situation. When a new situation is recognized by the client, it is sent to the server. Thereby, the situation can be a value of any kind of context. The server knows the corresponding *resource set* and sent this with all its included *resources* back to the client. This kind of providing and getting resources is called *context-aware data provisioning* [HSM16]. This makes it possible to cache only the resources that are needed in the current situation. When there are multiple active situations, all the resource sets will be loaded. On that way, the probability for a cache hit increases.

Based on this context-aware data provisioning it is possible to create selection strategies. These can be grouped in two different categories. The first category is *static context-*

aware caching where the resource sets depending on the context are fixed. The second is *dynamic context-aware caching*. Thereby, the content of the resource sets can change depending on computations on the server side.

Static Context-Aware Selection (SCAS) provides fixed resources sets. The included resources get defined and to not change until there is a conscious change by a user. When a situation derived by the value of a context element is recognized, the value corresponding resource set is loaded and cached by the client. For this purpose the server has multiple resource sets that include the defined resources. Creating a resource set that have information about a location, enables *location based selection* (CAS_1). Thereby a location can be a station or a building. This is explained with the help of the extract of the boat example and its logical data model. In figure 4.3 this extract is presented.

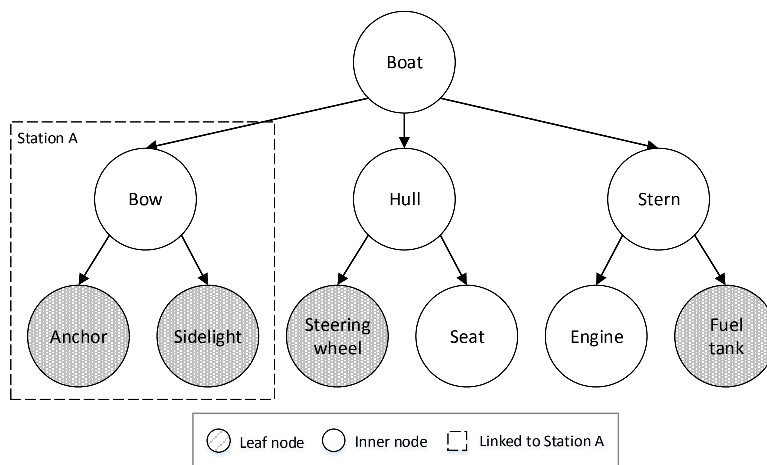

Figure 4.3: Context-aware selection - location based selection [HSM16]

The tree represents the product data of the boat. The components of the branch that is induced by *bow* are needed at *station A*. E.g. it is imaginable that the *bow* is produced at *station A* and information about the two components *sidelight and anchor* is needed. This information is included in the physical resources. On the server there exists a previously defined resource set that includes these resources. This is linked to *station A*. When the situation *station changed to station A* is recognized by the client, a message containing this location is sent to the server. As response, the server sends the

corresponding resource set. The resources get cached and can be accessed from the local storage the next time they are needed.

Role based selection is proceeded on the same way. Thereby, situations derived from the context element *role* are considered. A user can have different roles that are dedicated to its profile. On authentication the server check the roles of the user. If the server can found a resource set for the role, it sends it to the client where the user is logged in. This includes the resources that are linked to the role. This is useful because users with same roles often need to access the same resources. When the user changes the project or the responsibilities, the role can be adapted and the new linked resources will be cached.

Task based selection is also realized in this way. The situations that are derived from the context element *task* can occur on different ways. It is possible that the task is recognized by the movements of the user that executes a task. Another opportunity is that a user select a task from the task list in the client. Independent from the source which generates the situation, the server knows the corresponding resource set and sends the resources to the client where the resources will be cached.

Dynamic Context-Aware Caching (DCAC) reduces the time and effort spent on defining the resource sets. The resources that are linked to the context elements are determined automatically and can change over time. To make this possible, the context model from Hoos et al. [**Hoos2006**] is extended with an additional dimension *time*. Thereby time is an information, to which point in time a resource is requested from the server. This dimension can be combined with any other. It is possible to combine the location and the time to get the information which resource is requested at which station to which time. Therefore this dimension is related to each other. This information that relates the different dimensions with time, can be used to generate strategies depending on dynamic context-aware caching.

Most popular selection (CAS_4) based on a certain *role* is one of these dynamic context-aware strategies. The time is linked to the role and the accessed resource. On the basis of this data the most popular resources within a certain time interval 'T' is determined. For this purpose, all accesses to the resources from users owning a role are counted on the server. The resource set is determined with the following formula. Thereby, r is the role, t the time where the computation is executed, m the total numbers of resources and the absolute values the number of accesses to the resource.

(4.1) $RS_r(t) = max\{|R_1|, .., |R_m|\}$

In Figure 4.4 an example of this determination is shown. It is visualized, which resources that are stored on the server are accessed from users owning different roles in a specific interval.

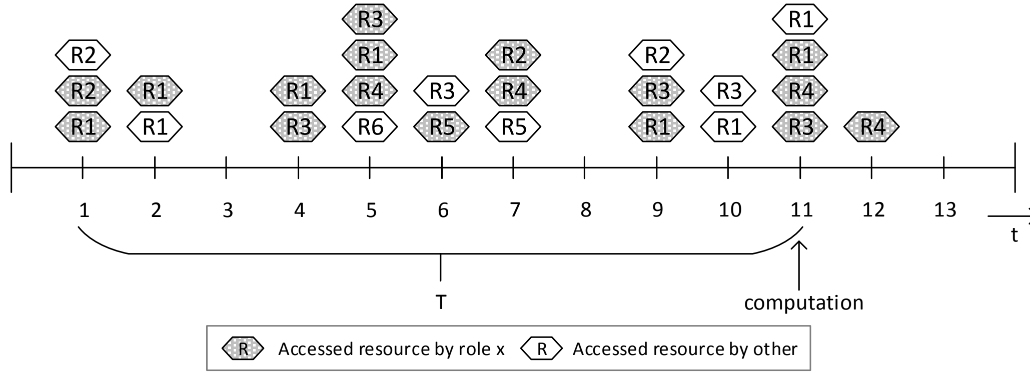

Figure 4.4: Dynamic Context-Aware Caching - most popular

The resources within the server can be accessed by a certain *role x* or by *any other role*. The time interval *T* the server goes back to count all the access done by *role x* is set to 10. It should be determined, which are the top 3 of this role. The result is computed as follows:

$$RS_x(11) = max\{|R_1|, |R_2|, |R_3|, |R_4|, |R_5|, |R_6|\} \Rightarrow max\{6, 2, 4, 3, 1, 0\}$$
$$\Rightarrow \{6, 4, 3\} = \{R_1, R_3, R_4\}$$

At time t = 11 where the *computation* is executed the most popular from role x are R_1, R_3 and R_4. This means that all users with role x get this resources from the server on authentication and cache these. When a new computation to another time is executed and new most popular arrive, the server will send these to the client too. It is also practicable to ignore the roles and determine the most popular in general. All clients independent from the logged in user will thereby get the most popular computed over all resource accesses in the time interval. In the example in Figure 4.4 the resources that will be sent to all clients are determined as follows:

$$RS(11) = max\{|R_1|, |R_2|, |R_3|, |R_4|, |R_5|, |R_6|\} \Rightarrow max\{9, 4, 6, 3, 2, 1\}$$
$$\Rightarrow \{9, 4, 6\} = \{R_1, R_2, R_3\}$$

4.2.2 Traditional Selection Strategy

In context-aware caching, the point in time may be reached, where all situations are processed on the client and the linked resources are cached. After this point in time it can happen that resources that are not linked to the processed situations are requested. They can't be found in the cache. As *traditional selection (TRS)* runs in parallel, these requested resources get also cached. This process is triggered by an explicit request of the user. If the resources are requested again, they can be accessed directly from the cache. On that way, frequently accessed resources get cached. When the maximum of the cache space is reached, it is the job of the replacement policy to decide, which resources get replaced.

4.3 Cache Replacement Policy

For the replacement policy a *gain based function with priority (GBFP)* is used. The priority is the parameter of the function. Through the different kind of selection strategies, the single resources in the cache come from different contexts. E.g. they can be part of a resource set that depends on a task, or they can be cached on traditional way. It is possible that at one moment, multiple situations are active (multiple context elements have a value). In this case, multiple resource sets are stored in the cache.

When the cache space limit is reached, the policy has to decide if resources in the cache should be replaced by new incoming resources. For this decision, the priority is used. The resource with the lowest priority in the cache gets replaced. For this reason every resource in the cache gets a priority on inserting. The priority of the resource starts always on inserting with the corresponding priority level. How the priority level of a single resource is set, depends on the use case of the app.

If the user of the app wants to be actively involved in the replacement process, it makes the most sense to give traditional selected resources the highest priority level. On that way, the last viewed resources are cached and won't get overwritten by context-aware selected resources. So the user can use the last viewed resources offline. If the user is only passively involved, context-aware selected resources should get a higher priority level. Thereby, the focus is more on a supporting functionality than on storing resources for an offline usage. The resources that will be needed in the context have a higher priority than the last viewed.

A recommended prioritization can be seen on table 4.1. The focus is on a supporting functionality. To give the users the opportunity to protect specific resources for

replacement, an offline snapshot is introduced. Thereby, all the resources that are currently visible for the user, get the highest priority when this function is executed. On that way the users can manage the resources they want to use offline.

Strategy	Priority level
Offline snapshot	60
Task based selection	50
Location based selection	40
Role based selection	30
Time based selection	20
Traditional selection	10

Table 4.1: Prioritization of the resources in the cache

With this values the gain function is set up. All the resources are marked with its priority level. When a resource set linked to a situation that is derived from *task* arrive at the client, all its resources get the priority '50'. If the cache space limit is reached the priority levels are used to replace resources. All resources in the cache that have a lower priority level than the new arriving, can be replaced. E.g. when *location* based resources arrive, all resources with priority level less than '40' are replacement candidates. Resources that arrive with the lowest priority level make an exception. As there is no lower level, resources with the same level can be replaced (instead of resources with a lower priority level). To select the resource within a priority level that should be replaced, an access frequency counter for each resource is introduced. This counter is incremented every point in time a resource is accessed. If the lowest priority level is '10' and the lowest access frequency of all resources with level '10' is '1', this resource will be replaced.

To prevent the overflow of the access frequency counter and to avoid that long staying resources will never be replaced (within a priority level), the counters will be reset when re-initializing the cache at the server. This occurs after a certain time the cache was last initialized. The priority level of still cached resources can be changed through a new situation. E.g. every context element on the client can get an empty value. This leads to the decrease of all the resources to the lowest priority level. Another possibility is that a resource gets a member of a resource set with higher priority. In this case, the priority level is increased.

The following formula describes the priority P of a resource, whereupon pl is the priority level, f the access frequency and x the resource identifier. The multiplier 1000 is used to give the priority level a higher weight.

(4.2) $P(x) = (pl_x * 1000 + f_x)$

The gain based function is composed as follows, whereupon t is the time the function is executed. This time is when a new resource should be inserted in the cache and no space is left.

(4.3) $GBFP(t) = min\{P(1), P(2), ..., P(x)\}$

With this function the resource with the lowest priority is determined. The priority level of this resource is compared with the priority level of the new resource that should be inserted in the cache. Only if the priority level of the resource is smaller (when new resource has lowest priority level smaller or equal) than the level of the new one, it gets replaced. May it is necessary to execute the function multiple times at one moment. E.g. when the size of the replaced resource is smaller than the size of the new one. In this case it has to be repeated until there is enough space left. To prevent that a too high access frequency will change the priority level, the maximum of a the counter is set to 9999.

An example of the replacement policy is described in figure 4.5. Thereby the cache of a client, the resources from the different context elements and the accesses to them are shown over time.

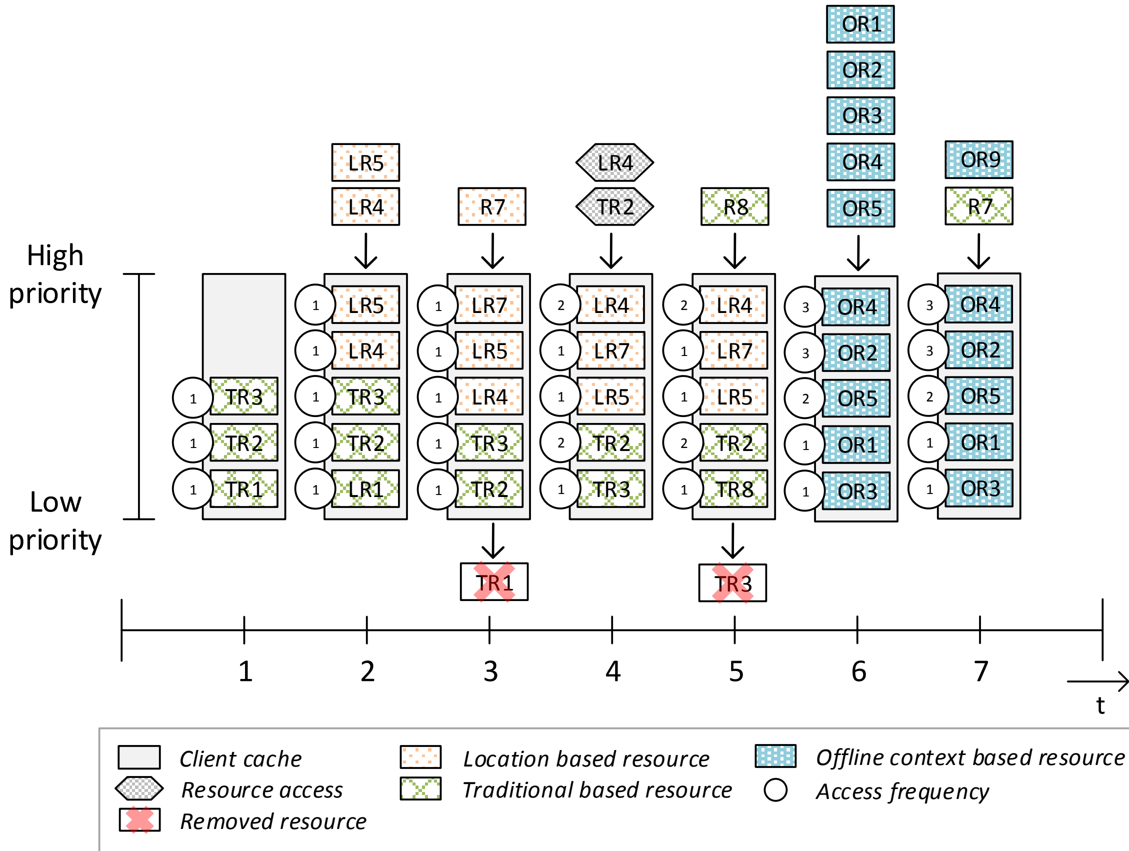

Figure 4.5: Gain function based replacement strategy

The resources in the *client cache* are ordered descending after the priority. The resource with the highest priority is located on the top and the one with the lowest on the bottom. At time t = 1 the cache of the client includes the resources TR_3, TR_2 and TR_1. Later a resource set that contains the *location based resources* LR_4 and LR_5 arrive at the client. As there is enough space left in the cache, the two resources can be inserted without replacing other resources in the cache. All the resources start with an access frequency counter of '1' after inserting. At time t = 3 more resources arrive, but at this time there is not enough space left and the client have to replace resources. To compute the resource that will be replaced the gain function is used. At this moment two different priority levels are in the cache. As the *traditional based resources* have a lower priority level, a resources from these will be replaced. This is computed as follows:

The result of the function at time t = 3 is that resource TR_1 with a value of '100001' should be replaced. This represents the lowest priority. TR_2 and TR_3 have the same priority, but in this case the first occurring resource of those are chosen. As the priority level of the new incoming resource is higher (40) than from TR_1 (10), the replacement is executed. After that, LR_7 can be stored in the cache. By accessing a resource from the cache, the access frequency counter will be incremented. This can be seen at time t = 4. Here the resources TR_2 and LR_4 are accessed, which increments the counter. This changes the priorities within the cache. Now resource TR_3 has the lowest priority. At time t = 5 this resource is replaced by TR_8.

Through a resource set from *task* that is marked as offline context, the whole cache will be needed. This occurs at time t = 6. Here two different situations are active at the same time (*location* and *task marked as offline context*. As the *offline context based resources* have a higher priority than all others, all the resources in the cache will be replaced. Some of the arriving resources are still in the cache. These won't be replaced. Instead, only the priority level changes and the counter is incremented. As the whole cache is filled with resources with the highest priority, no one will be replaced by an other arriving resource. That is important to guarantee that all the needed resources are still there when being offline. Only when the priority level of the resources change through a changing situation (e.g., offline mode is finished) other resources can be cached.

4.4 Cache Coherence Strategy

To invalidate obsolete resources in the cache of the client, an invalidation protocol called *Stateful Invalidation Protocol (SIP)* is used in the caching mechanism of the concept. On mobile devices there is a need for strategies that do not produce too much traffic. Client Polling is therefore not suitable. Every time a resource is accessed, the server is asked if the resource is up-to-date. This will lead to a lot of unnecessary requests. In this regard Time-to-Live would perform better. A longer lifetime of the resources in the cache would produce less traffic. The problem here is that under certain circumstances the clients have to work with obsolete resources for the whole lifetime. This is not preferable in the engineering domain. If resources are accessed when being online, it is expected that they are up-to-date. Otherwise, the users can never be sure, if they use the newest version of a resource. Therefore, there is a need for strong consistency during the online time.

For this reasons a stateful strategy that allows sending selective invalidation reports to the clients is the most suitable. The server remembers all the resources a client has

cached. When there is a new version for a resource, the server checks which clients have cached this resource. When there is found a client and it is also connected, the server sends the invalidation report that includes the obsolete resource. On this way no unnecessary messages are sent and the clients are informed about obsolete resources as soon as possible. It is possible to guarantee strong consistency when being online. A detailed process of this invalidation protocol is presented in figure 4.6. Thereby, the client cache and the remembered cached resources from this client on the server are shown over time.

Figure 4.6: Stateful Invalidation Protocol (SIP)

The first step the client has to do, is the initialization at the server. At this point in time the client sends its cached resources (R1,R2) and the timestamps of them to the server. The server stores these information. At time t = 2 a new version of R1 arrive at the server. So the client has an *obsolete resource*. The server knows that the client has cached this resource and with the help of the timestamp it is identified as *inconsistent*. The server also knows that the client is connected an sends the *invalidation report*

with resource R1 to the client. When the client receives this report, it deletes the old version of the resource, gets the new and *acknowledges the receiving*. So the server knows at time t = 3 that the cache is in a consistent state again. After that, the client *requests the resource* R3 from the server. This is remembered by the server. After time t = 4 the client *gets offline* until time t = 7. During this time, new versions of resource R1 and R2 arrive at the server. As the server knows that the client is not connected, no *invalidation report* is sent. When the client gets online, the server knows that and sends the *invalidation report* with R1 und R2 to the client. At time t = 8 the consistent state is reached.

When the cache space limit is reached and new resources are requested, it may happen that resources *get replaced*. This occurs at time t = 9, as the client has *request resource* R4 and R5. There is not enough space left to store both in the cache. As it is the task of the replacement policy that runs on the client, the server does not know which resource will be replaced. It would produce too much traffic informing the server about every replacement. Therefore, this is not performed. The server will still keep this resource remembered and *send invalidation reports*, if the resources is affected by an update. Like at time t = 10 the client responds on a falsely sent report, with an acknowledge that contains the information that the resource is not in the cache. So the server can forget the resource. Now it can happen that a lot of resources get replaced on the client. At the same time, no updates are performed. So the list on the server gets bigger and bigger. To avoid that it gets too big, after a certain time window 'w', the client has to *initialize* again. After that, the client and the server are back in sync. How 'w' is chosen depends on the hardware of the server and the number of clients. If the server has a lot of power and a manageable amount of clients, the 'w' can be chosen to be multiple weeks.

4.5 Conclusion

The described strategies compose the caching mechanism of the concept. For summarizing the used strategies Figure 4.7 is used. It is presented, which strategies are used in the single categories of the caching mechanism.

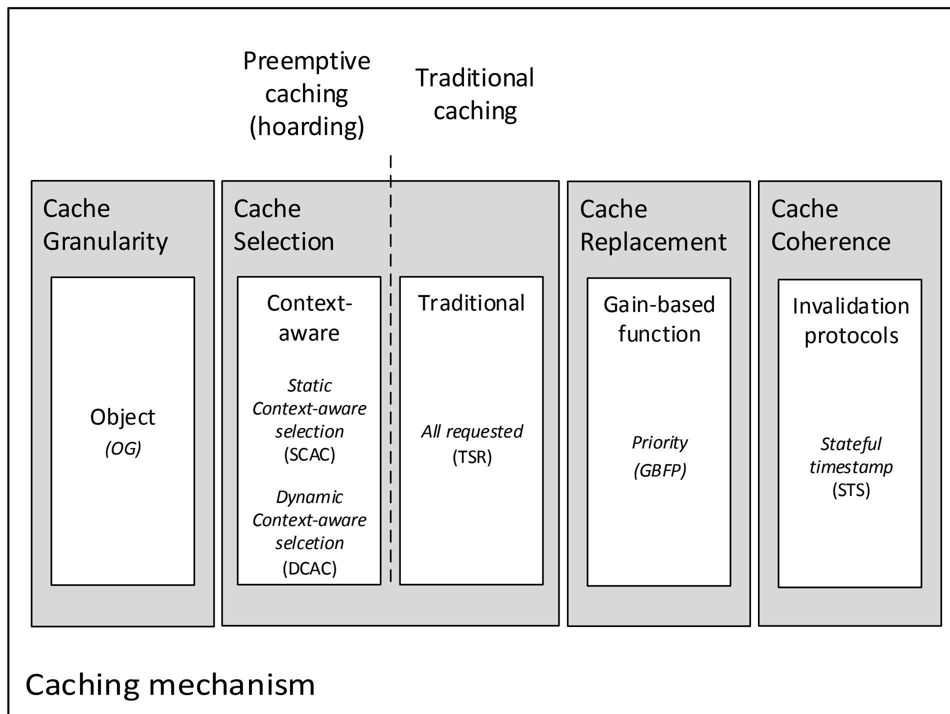

Figure 4.7: Overview of the caching concept

The chosen *Cache Granularity* is objects. Theses are represented by the single resources. That means that the smallest unit in the cache is a resource. Within the concept both *traditional caching* and *preemptive caching (hoarding)* techniques are used. For *preemptive caching (hoarding)* the *context-aware selection strategies* are used. It is distinguished between *static context-aware selection* where the resource sets are fixed and the *dynamic context-aware selection* where the resource sets can change over time. Within the strategies different context elements are supported by the concept (task, location, role, time). *Traditional caching* is realized through the *traditional selection*, where all the requested resources are cached on the device. The different selection strategies can be used in parallel. This is possible through the use of the *gain-based function* that uses priorities to replace resources. On that way resources that are needed in the current situation are not overwritten by others. The relevant resources get cached that leads to a good utilization of the available cache space that increases the cache hit ratio. To invalidate obsolete resources in the cache an *invalidation*

protocol is used. The *Stateful Invalidation Protocol* allows to selective sending the invalidation reports without producing too much overhead.

5 Prototype

In the following chapter, the realization of the created caching concept within a prototype is described. It is explained, how the architecture of the existing app is changed, to enable a caching mechanism. The different software components of the new architecture are described by detail. Also the used technologies are presented. At the end of this chapter, the protocols and processes that are needed to realize the different strategies of the caching mechanism are explained.

5.1 The Existing App

For the implementation of the caching concept a mobile app that is used in the engineering domain exists. Azevedo et al. create a portable and platform independent viewer that allows to visualize and examine 3D models of products [AKL]. These 3D models come from PDM systems. It is optimized for the usage on mobile devices. It allows to use gestures like panning, swiping and zooming for interaction. To get a better overview, the model is transformed into a tree. This can be used for navigating through the model and for selecting specific components. For the integration of the caching mechanism it is important to know which architecture is used and which data model has to be considered.

5.1.1 Existing Architecture and Used Technologies

As the app should run platform independent, web technologies are used. HTML in combination with CSS and Javascript makes it possible to run the app with ordinary web browsers on nearly any device. For showing the 3D model, the JavaScript library Three.js [1] is used. This uses WebGL for rendering. The tree viewer which serves as an overview over the model, is based on a SVG file. SVG can easily be manipulated with

[1] http://threejs.org/

JavaScript and enables interactions on this way. For the creation of the SVG D3.js [2] is used. The architecture is shown in figure 5.1.

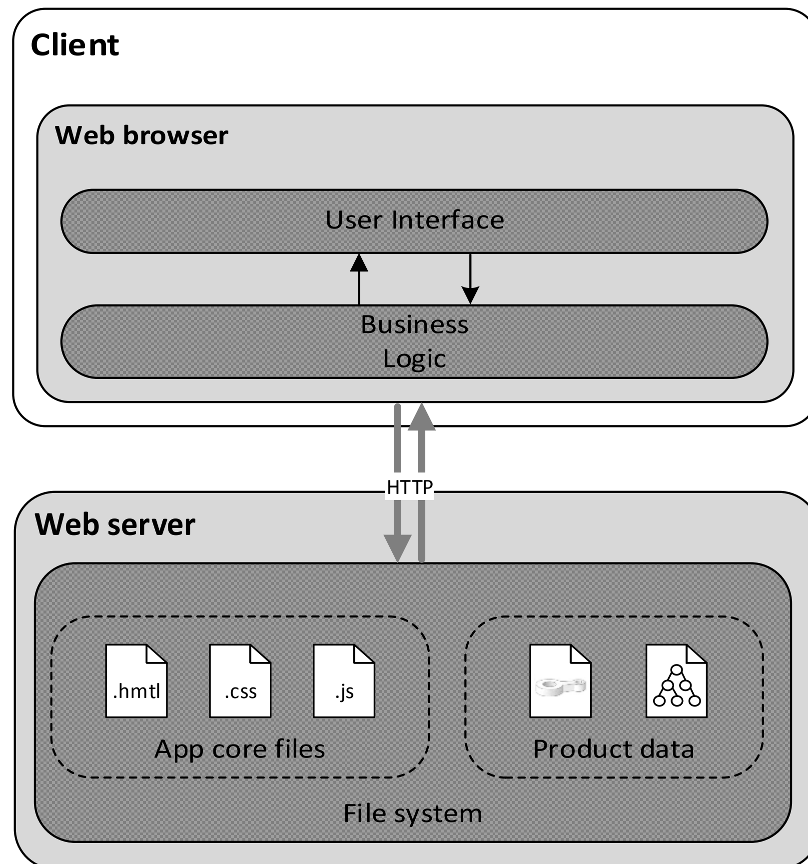

Figure 5.1: Architecture of the existing app

The app is executed in a *web browser* on the client. The *core files of the app (HTML, CSS, JavaScript)* that are needed to run the app are located on the *file system of the web server*. These files compose the two software components of the app. The *User Interface* enables the interaction with the model and dispatch the commands from the user to the *Business Logic*. The *product data* that is also stored on the *file system of the web server* are needed from the *Business Logic*. They are used to process and visualize the model of the product in the viewer. These have to be requested over *HTTP* every time a model is accessed by a user (if they are not temporary cached by the browser).

[2] https://d3js.org/

5.1.2 Data model

The data model that is used in the existing app, is also used for the prototype. It consists of the product data that is needed for rendering the model. Logical it is represented by a tree, where the single components of the product are the different leaf nodes of the tree. Physically the nodes are mapped to JSON files. The visual presentation of this is shown in figure 5.2. Here the mapping of the logical to the physical data model is presented.

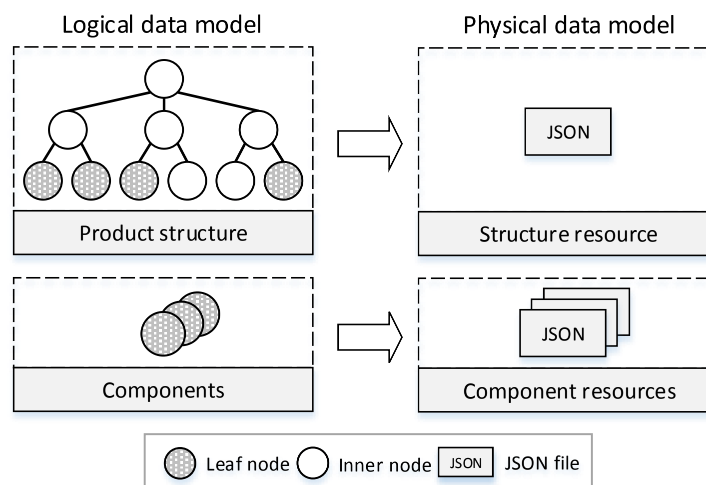

Figure 5.2: Data model of the protoype

Every component of the product that is represented by a leaf node, has its own JSON file. The file contains different information that is structured in form of attributes. They contain meta data like the name and the version of the component. Besides these meta data, attributes containing the geometry data are included. These are important for rendering the model within the viewer. For the structure of the tree of one product, a separate file exists. This is also in JSON format. In this file all the other JSON files are referenced by the name in a nested structure. This allows to reproduce the tree.

The base of these JSON files is composed of PLMXML. PLMXML is a common industry standard that is used for a product specific data exchange [Com05]. Thereby, information about the product structure, shape information and process information are included in a PLMXML file [DBM+07]. To extract the information (meta data and geometry data) that is needed to visualize the 3D models in the viewer, a convert is

used. Azevedo et al. provides a converter, that uses PLMXML files and convert them to JSON [AKL].

5.2 Overall Architecture

In the existing architecture, there is missing a communication mechanism that allows to get the different product data dynamically from a server. It is only possible to take all the JSON files of the product data and put them directly on the file system of the web server. Here they can be accessed from the app. This is not very practicable. It is better to have a back-end component that allows the app to manage the product data that come from the PDM-Systems. On that way it is possible to provide all available models to the clients. Also different versions of models and their components can be managed.

For this reason a back-end component on an application server is implemented. It provides some basic interfaces that can be used for getting the product data. These interfaces are also used to fill the cache on the client side with the specific resources. For using the interfaces and for enabling a caching, some additional software components and protocols are needed. The extended architecture can be seen in figure 5.3. Within the architecture, the old and new components can be distinguished. It is also shown, where the cache is integrated into the app.

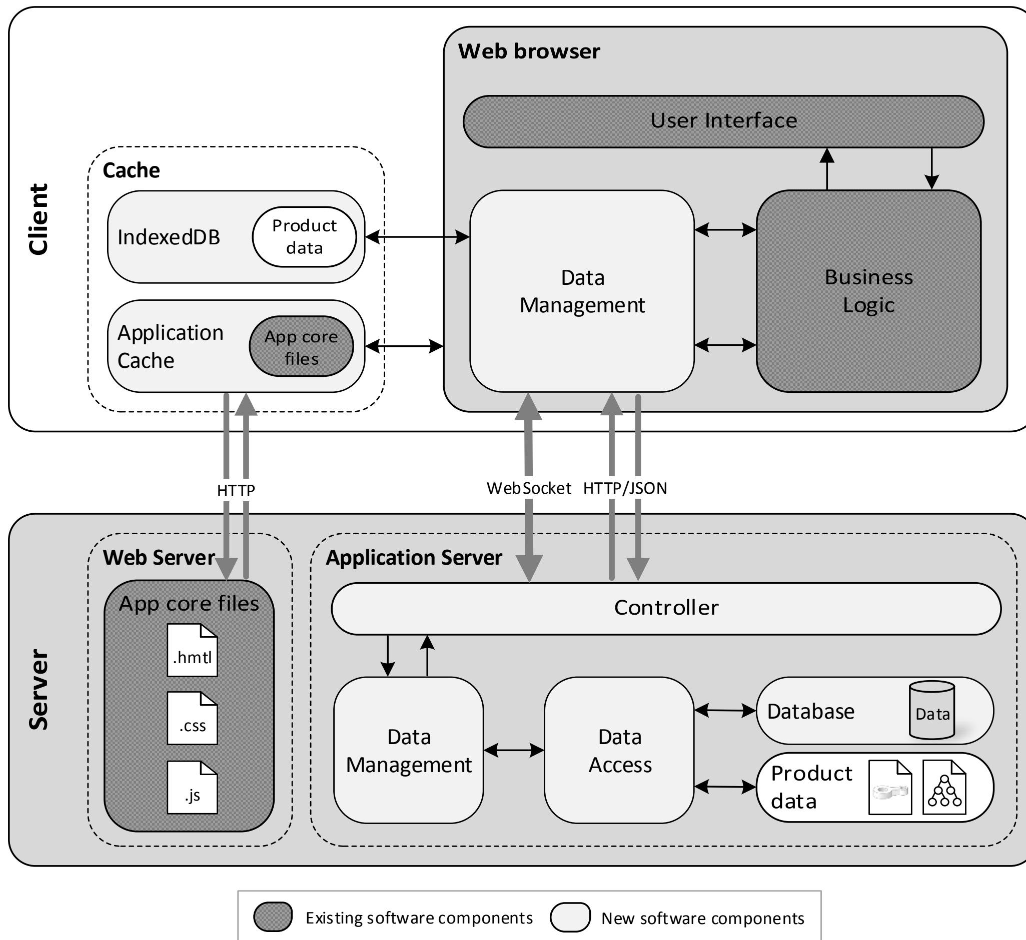

Figure 5.3: Architecture of the prototype

The two tiers of the app are the *client* and the *server* tier. The *client* tier contains the old software components that are necessary for rendering and interacting with the model(*User Interface and Business Logic*). These are still executed within a browser on the *client*. *The cache* that consists of two different local storage's is located on the *client* side. Thereby, the *Application Cache* includes all the *core files of the app* that are needed to run the app (*JavaScript, HTML, CSS*). These files are requested from the *web server* when the *client* opens the app for the first time. After that, these files get cached and the app can be executed without connection to the *web server*.

The second storage is the *IndexedDB*. Here the resources are cached and hoarded (*product data*). The *data management* forms together with the old software components (*User Interface and Business Logic*) the core of the app. It is responsible for getting

the resources that are needed for rendering the model and generating the tree. The component can communicate with the *IndexedDB* and with the *server*. When the resources that are needed can't be found in the *IndexedDB*, they are requested from the *server* (if a connection exists). When there is a need to cache resources, the *Data Management* component inserts them into the *IndexedDB*.

For the communication between the *client* and the *application server* two different protocols are used. The first protocol is for requesting and transferring the resources. Therefore *HTTP requests* that are realized over a REST API are used. The media type for the resource exchange is JSON. The second protocol uses a *WebSocket* for communication. This is needed for the invalidation of the obsolete resources in the cache (*IndexedDB*). On the *server* side the *Controller* component is responsible for receiving, transferring and dispatching the messages that are sent over the protocols. The *Data Management* component of the *server*, has the task to get the resource sets that depend on the context. It has to provide the resources and has to determine the invalidation reports that have to be sent to the *clients*. For these purposes it has to communicate with the *Data Access* component that includes all queries needed for getting the resources. The *database* contains the links to the JSON files on the *server* that includes the *product data*.

5.2.1 Used Technologies

For implementing the new software components that are needed for the caching mechanism, different technologies are used. On the client side, web technologies are used (HTML, CSS, JavaScript). For the back-end component Java is used. In the following Section a short overview of the used technologies within the prototype is presented. The details of the usage of these components are explained after this overview.

- **Client components**
  For the implementation of the new software components of the client mostly jQuery [3] is used. It makes it possible to use AJAX for requesting the data from a server. With the JavaScript library STOMP.js [4] it is possible to establish a *WebSocket* connection. This enables a bidirectional connection between the server and the client. Both client and server can send messages.

[3] https://jquery.com/
[4] hhttps://github.com/jmesnil/stomp-websocket

- **Application Cache**
The *Application Cache* is part of the HTML5 Offline Storage's. The intend of this cache is to store a copy of the web app. All the core files of an app can be stored there and accessed without an internet connection. The files that should be cached are defined in a manifest file. The manifest file is located on the web server and linked within the HTML-tag of the start page. All other sub pages of the app must also have the reference to the manifest file in their HTML-tags. A manifest file on the server must be from type '.appcache' [KH08].

- **IndexedDB**
Also a component of the HTML5 Offline Storage. The data is stored in a key-value format. It is possible to create different stores within a *IndexedDB* that act like tables in relational databases. Each store has it's own key that is used for the access to the data. Besides the keys, different indexes can be created to locate the data. The communication with the *IndexedDB* is realized over the HTML5 Offline API [MSG+15].

- **Server components**
The server components are created with the Java framework Spring Boot [5]. With this framework it is possible to create a REST API with minimal effort. It provides also a *WebSocket* that can be used for the bidirectional connection. For converting to JSON the framework uses Jackson. As a Tomcat application server is integrated, it is possible to create a single jar file that can be executed as a simple Java Application. There is no need for an external application server.

- **Database (SQLite)** For the database *SQlite* [6] is used. It is an embedded SQL database engine that need no separate server process. The reads and writes are executed directly to an ordinary file on the disk. It is possible to set up a database with minimal effort.

5.2.2 The Cache

In this subsection the cache is explained by detail. It is composed of the *IndexedDB* and the *Application Cache*. Within the cache, it is distinguished between the core files of the app and the resources (product data). The core files are needed to run the app. The resources instead are needed from the *business logic* to process and view the 3D models. For both a separate storage is used. How the two storage's are used within the

[5] http://projects.spring.io/spring-boot/
[6] https://www.sqlite.org/

prototype is shown in Figure 5.4. With the help of this Figure, both the storage's are explained by detail. It is described, which resources are cached and how they work.

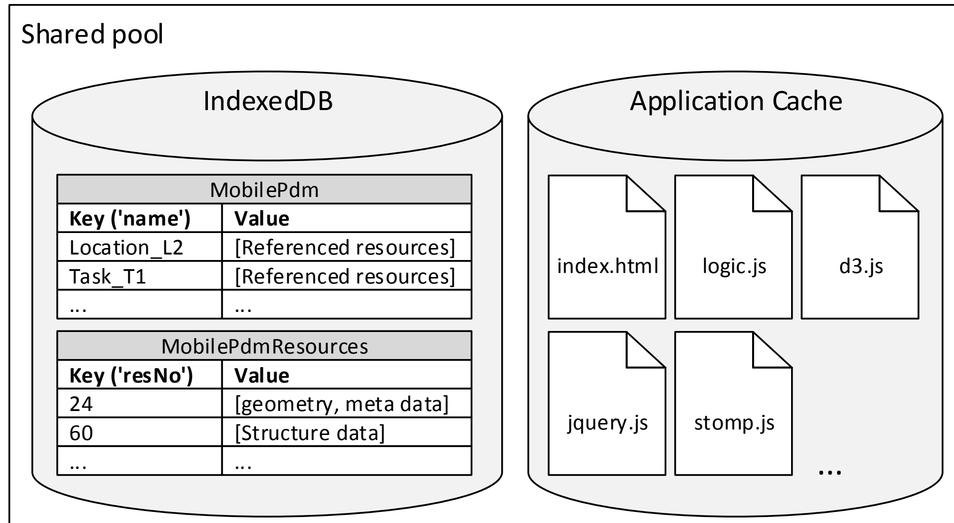

Figure 5.4: Cache of the prototype

Application Cache

The *Application Cache* is used to store the core files of the app on the device. This includes all the files that are part of the *user interface, the business logic and the data management*. To enable the caching, a manifest file containing all these files is created. This file is stored in the root folder of the app on the web server and linked within the HTML-tag of the start page (index.html). An excerpt of this manifest file can be seen in listing 5.1

Listing 5.1 Manifest file for caching the core data of the app

```
CACHE MANIFEST
# v1
CACHE:
index.html
js/custom/custom.js
...
NETWORK:
*
```

The manifest file starts with *CACHE MANIFEST*. Lines with # are interpreted as comments. This is used as a version number of the app. When there is a new version of the app, the number is incremented and all the clients will request the new version with all its files. Changes to the file lead to a renew of the cache at the client. Under the header *CACHE:* all the files are listed that have to be cached. Therefore the paths to the file on the server are used. The *NETWORK:* header is used to define resources that have to be explicitly requested from the server. The wild card * means that all files that are not listed under *CACHE* are affected.

When the client requests the start page (index.html) for the first time from the server, all the referenced files will be loaded. On receiving the files are stored in the *Application Cache* from the client. Now they can directly be accessed from the device. There is no need to communicate with the server. On this way the core files of the app can be used offline.

IndexedDB

The second component of the cache is composed by the *IndexedDB*. Here all the resources are stored that are selected by the different selection strategies of the caching mechanism. Besides the resources, the resource sets from the active situations are stored. This is necessary to set the priorities of the resources within the cache correctly. As both of them are identified by different keys, there is a need to split them in two separate stores within the *IndexedDB*.

The first store is for saving the resources that represents the components of the product and the product structure (MobilePdmStoreResources). Thereby, the key of the store is the resource number (*resNo*). The value contains the data of the corresponding JSON file (geometry data, meta data, structure).

In the second store (MobilePdm), the resource sets from the active context are cached. Here the name (*'name'*) is used as the key. To distinguish between the different kind of context elements the name of the resource sets includes a prefix (task_, location_, role_). This is important for setting the priorities. Within the values of the resource sets, the references to the resources in the cache are stored. They are referenced with the resource number (*resNo*.

5.2.3 Database

In the database all the data that is needed for the caching mechanism is stored in a single file that represents the database (MobilePdm.db). The enhanced entity - relationship (EER) model can be seen in figure 5.5. In the following Section, the usage of the single tables of the database will be explained.

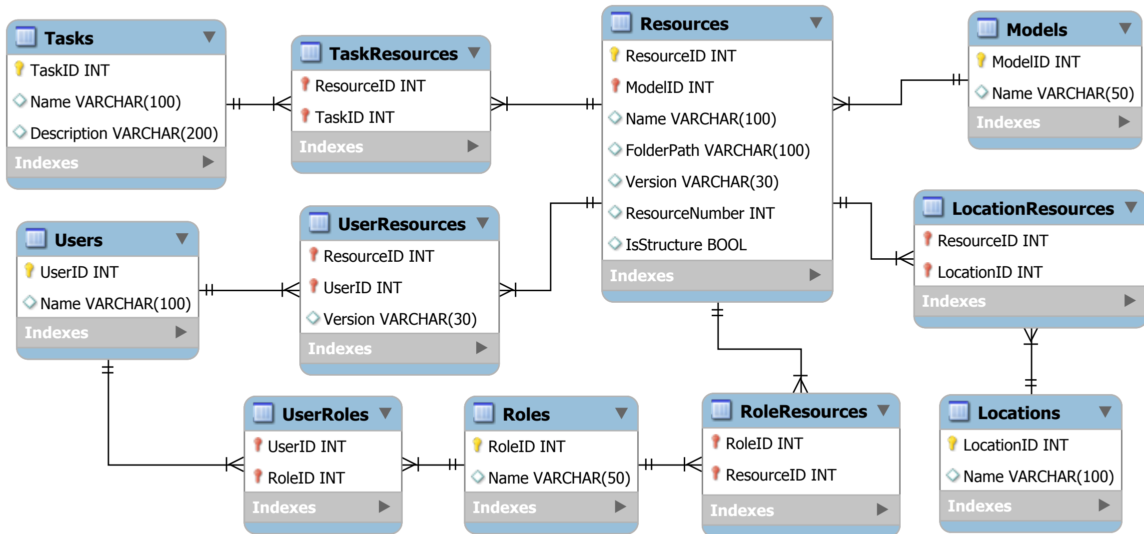

Figure 5.5: EER model of the database

- **Resources**
  A resource that is represented by this table, can either be a component, or the structure of a model. With the flag IsStructure it is distinguished between them. The *FolderPath* determines the path, where the corresponding JSON file can be found on the server. With *Name and ResourceNumber* the resource can be identified. With the help of the field *Version* it can be determined, which version of the resource is stored on the server. This is important for the invalidation process of obsolete resources. By means of the foreign key *ModelID* it is possible to assign a resource to a model. As the geometry data of a component resource includes the exact coordinates of the position within a model, it is unlikely that it is used in multiple models. Therefore this 1:n relation is chosen.

- **Models**
  With this table it is possible to group all the resources to a product model. It is recommended that a model has always an assigned structure resource. Otherwise it won't be possible to navigate through the model within the app.

- **Tasks/TaskResources, Locations/LocationResources, Roles/RoleResources**
  These tables are formed on the same way. They include information about the context elements and the the resource sets. The context element tables (Tasks, Locations, Roles) includes the name of the particular situation. Hence, *Name* represents the value of the context elements. To distinguish on the client between the different context elements, the values starts with a prefix ('task_', 'location_', 'role_'). The linking tables (TaskResources, LocationResources, RoleResources) includes the links between the situations (context element values) and the corresponding resources. They represents the resource sets. Within a resource set, it makes sense to link always the structure resource that corresponds to the model of the resources. They are needed in conjunction.

- **User**
  In this table all the registered users are stored with name. The identification of the clients is realized over the user. If a client subscribe by the server, a user authentication is required. On this way it is possible to trace back the session to the client.

- **UserRoles**
  A user can have different roles. This is realized over this linking table. On the basis of the assigned roles, it is possible to identify the single corresponding resource sets. When a user authenticate at the server with role x and there exists a resources set (there are linked resources to x), it is sent to the client where the user is logged in.

- **UserResources**
  In this table all the resources are stored that a client has requested (mapping to client is realized over the user). With the help of it, the server knows all the resources, a client had cached (stateful). On initialization of a client cache at the server, all client corresponding resources are deleted from here. Afterward, all the resources in the cache from the initializing client are stored. Every time new resources are requested from the client, the table will be updated. Thereby, the version of the resources that is sent is stored. This table is used to create the invalidation reports for the clients.

5.2.4 Priority List

Within the *data management* component of the client, there is an important list that is needed for the realization of the replacement policy. This list includes different objects that contain the information about the priorities of the single resources within

the cache. Therefore it is called priority list. With the help of this list, the gain based function with priorities (GBFP) is realized. It is much easier and faster to manage the priorities in a list, instead of iterating directly through all resources in the cache. This would take a lot of more time to determine the resource with the lowest priority. In table **??** an example of the used priority list is presented. It is shown, how the objects in the list are structured.

resourceNumber	**size**	**priority**
0	2000 KB	99999
3	500 KB	200052
16	1250 KB	10035
29	250 KB	20001

Table 5.1: Priority list

Every object in the list has three attributes. The *resourceNumber* is for identifying the resource. The second attribute includes the *size* of the object. This is necessary to calculate the free space in the cache. The *priority* of a resource is stored in the third attribute. It is calculated with the equation 4.2 that includes the priority level and the access frequency. Within this list it is possible to add new entries and update existing. E.g. updating is necessary when a resource is accessed and the access frequency increments (priority changes). Also if the membership to a resource set changes, it may be necessary to adapt the *priority* When there is a need to replace resources, the list is sorted descending after the *priority*. Before the last entry of the list is deleted, its *resourceNumber* is used to delete the resource in the cache. The first entry in the list has a special function. It includes the total size of all the resources in the cache. It can be identified by means of '0' as *resourceNumber*. As this entry should not be replaced, it has the highest possible *priority*. To prevent that the list gets lost through shutting down the app, the list is stored in the cache (MobilePdm store in the IndexedDB).

5.3 Implementation of the Caching Concept

In the following Section the implementation of the caching mechanism within the prototype is described. A caching mechanism is only needed for the resources (product data). For the core files of the app, there is no need. They are fix and only changes when the app gets updated. The files are small enough, so it is possible to store all of them in the Application Cache. So it is not necessary to define several strategies. For this reason the caching mechanism only considers the resources (product data).

Besides the software components and the storage's (cache and database) provided by the architecture, there is a need for logic and communication protocols to realize the mechanism. To make it clear, at which position the logic and the protocols are used, first the process of showing specific components of a product within the prototype is illustrated in figure 5.6. Thereby, it can also be seen, on which way an offline mode is possible. The focus of this illustration is on proceedings needed for the caching mechanism. So not all processes needed for showing the components within the prototype are included. For a clearer view, only the references to the protocols are given. These are explained later in detail.

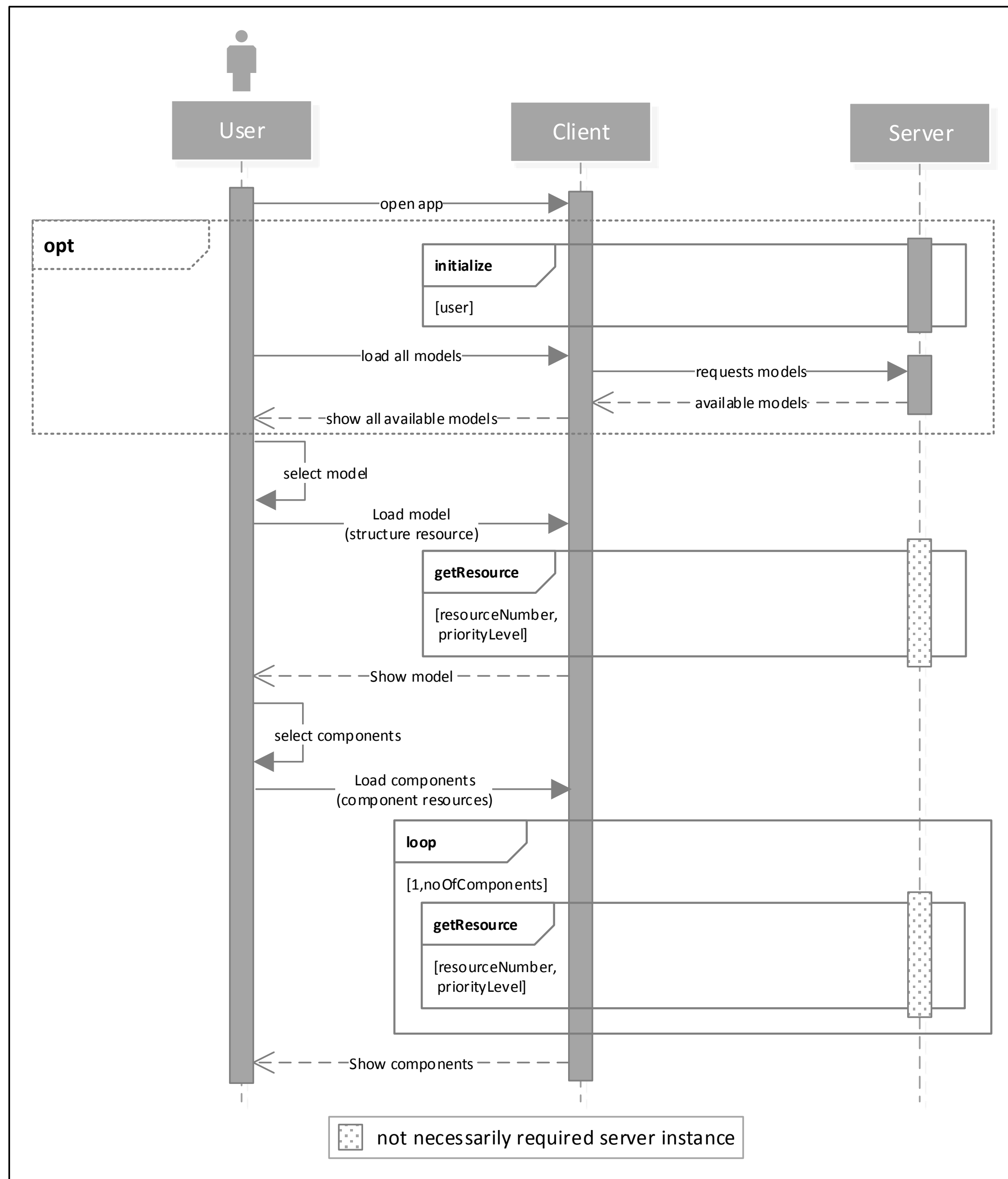

Figure 5.6: Process for showing product components

The process starts with *opening the app*. The then following block is optional (*opt*). This means that this part is not mandatory needed to work with the app. Through the *initialize* protocol, the client authenticates at the server and create a WebSocket connection. This is essential for the invalidation process. As it should be possible to work offline with the app, this is an optional step. Also the *loading of all models* from

the server is not necessary. If there are still models in the cache (structure resources), these can be used. Only if the user want to access a model that is not cached, this step has to be executed.

The first needed step after *opening the app* is the *selection of a model*. After this selection, the client has to *load the structure resource of the model*. This is realized over the *getResource* protocol that needs the *resourceNumber and priorityLevel* as parameter. The *resourceNumber* is needed for identifying the resources and the *priortiyLevel* for caching and replacing them. This protocol can also be executed offline. The server is not mandatory needed. If the resources that are needed can be found in the cache, it is possible to get them directly from here. There are two ways (techniques) of caching resources supported by the implemented mechanism. Traditional and preemptive caching through static context-aware caching. These are realized through two other protocols that will later be explained in this Section.

When the structure of the product model is available for the client, the user can *select components* that he want to see. *Loading the component resources* is also realized over the *getResource* protocol. When the resources can't be found in the cache, they are requested from the server. After requesting the resources they may have to be cached. Therefore, the replacement policy is triggered every time a resource is received from the server.

5.3.1 Cache Granularity

The data model of the existing app consists of product data that includes the structure of the product model (tree) and the components. Both are represented as JSON files. Within the JSON file of a structure, all the sub components of the model and the arrangement of them are stored. Therefore this file is needed as a whole. The JSON files of the components contains the geometry data that consists of different attributes needed for rendering (vertices, colors, faces, etc.). They are also needed in conjunction. If there is missing one attribute, the component can't be visualized correctly. For this reason the resources that are cached, includes the data from the JSON files. One resource represents one JSON file. Thereby it is distinguished between structure resources and component resources. Both contain different kind of information. But in the cache, both are treated identical.

5.3.2 Cache Selection Strategy

Both traditional and preemptive selection strategies are implemented in the prototype. For preemptive selection, static context-aware strategies are used (task, location and role). All are processed on the same way. Besides this preemptive selection, all requested resources will also be cached. When caching resources on this traditional way, they get the lowest priority in the cache. That means that when the cache is full with resources that are included in the resource sets from the context-aware strategies, the traditional selected will not be cached. This is managed by the replacement policy. To enable traditional and preemptive selection strategies, different protocols are used.

Traditional selection strategy Traditional selection is always active. When a resource is requested that presumes that it is not already cached, it gets cached by the client. Therefore, the protocol *getResource* is used. Traditional selection is triggered from users, when they want to see a model or specific parts of it. In figure 5.7 this protocol is presented. A sequence diagram that shows the interaction between the client and the server is used.

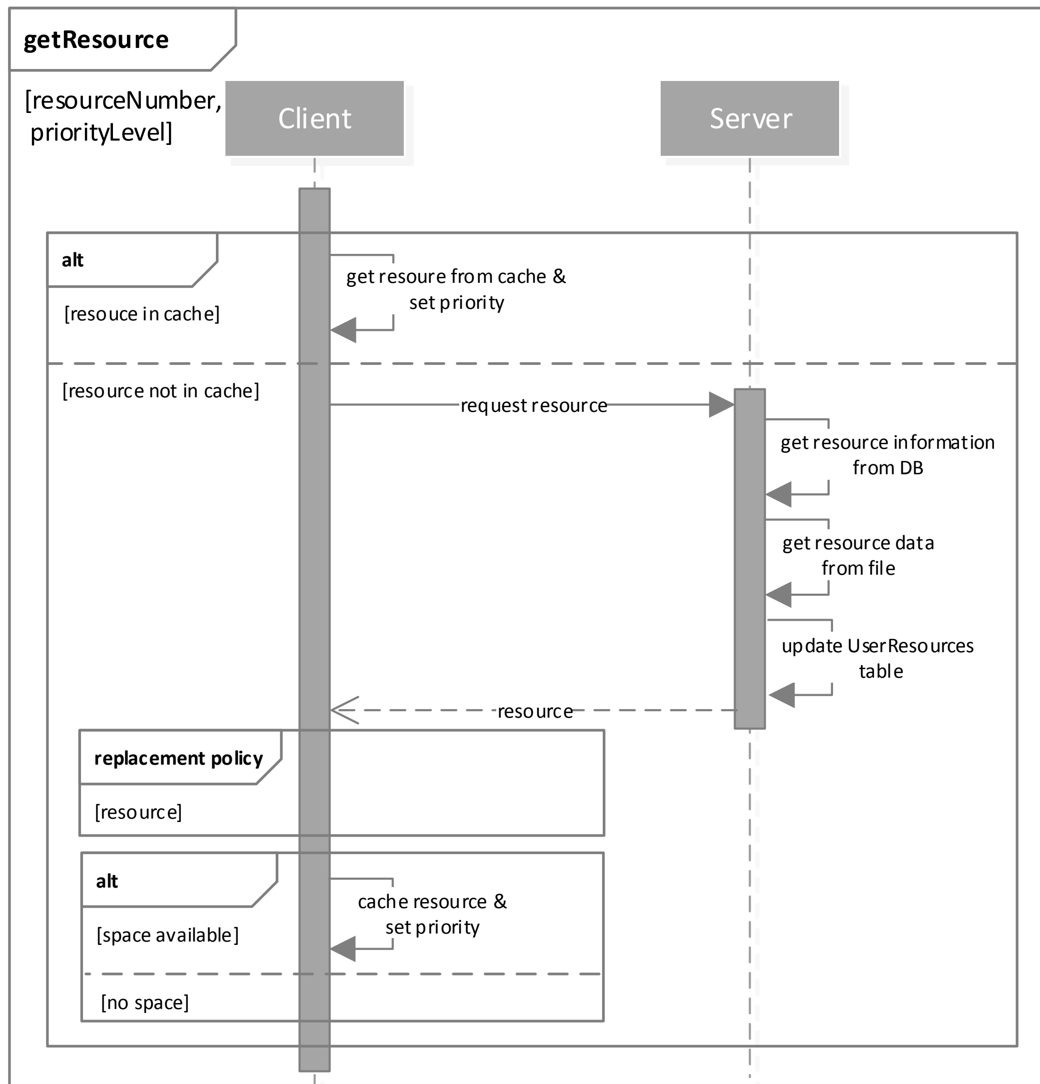

Figure 5.7: Protocol - getResource

The protocol needs two parameter as input. The *resourceNumber* is used to identify the resources on the server. The client has the *resourceNumber* information from the model and the corresponding structure resource. To set the correct *priorityLevel* of a resource that will be cached the second parameter is used. In the case of traditional selection, all the resources get the lowest priority level ('10'). As the *getResource* protocol is also used in the context-aware selection, the *priorityLevel* has to be a parameter. For determining the priortiy level, the table 4.1 from Section 4.3 of the caching concept is used.

When the resource is requested, there are two alternatives (*alt*). The first is that the *resources can be found in the cache*. In this case, only *the priority is set* (if necessary) and the resource can be delivered. The *priority is set* on this way that either the access frequency raised, or the *priorityLevel* changed. This depends on the selection strategy that executes the protocol.

If the *resource is not in the cache*, it has to be *requested* from the server. Therefore, the *resourceNumber* is sent to the server. The server uses the *resourceNumber* to *get all the resource information* from the database (name, version, folderPath, etc.). Then the folderPath is used to *get the resource data from the JSON file* (geometry and meta data). An *entry in the UserResource table is created* that the client has cached this resource. Before the client can cache the resource on receiving, the replacement policy is executed. The *resource* with the *priority* as attribute, is sent as parameter. Only if the replacement policy can guarantee space for the new resource (depends on size limit and priority), the resource can be cached. When caching the resource the *priority* is set. With this protocol, all the resources that are requested will be cached. This represents the traditional selection strategy that enables traditional caching.

Context-aware selection strategy To enable a context-aware selection of resources, a mechanism for the recognition of new situations is necessary. A new situation is a changing value of the context elements that are considered. Within the prototype these are task, location and role. As the prototype is not connected to any sensors of the mobile device, the situations are simulated. When connecting to the server, the app will get all available situations. They are listed in the app and can be activated. On activation, the resource set is requested with the situation as parameter. The process of requesting a resource set can be seen in figure 5.8. The interactions between the client and the server is shown.

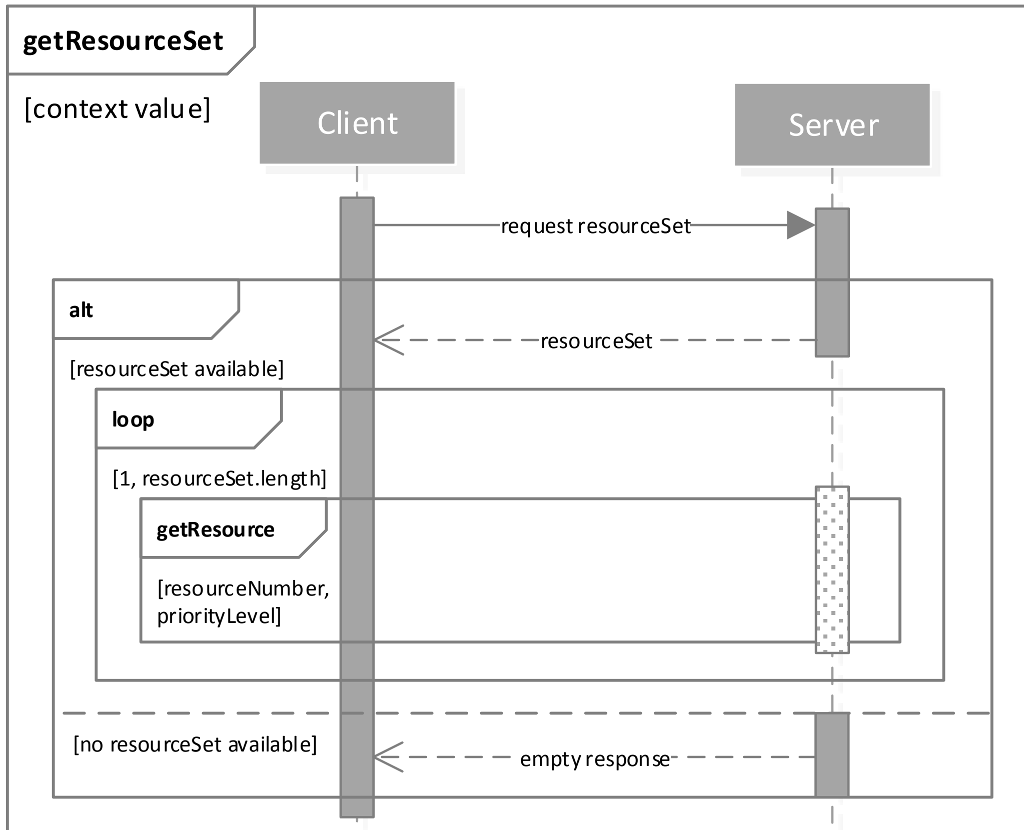

Figure 5.8: Protocol - getResourceSet

To *request a resource set* from the server, a connection is necessary. To perform the request, the client sends the context value of the simulated situation to the server. The server searches in the corresponding tables for an entry. E.g. when the context value has a prefix 'task_', the server searches in the Tasks table of the database. If the server can found an entry, it will create the resource set with its linked resources (linked in table TasksResources), and send it back to the client. Every resource in the set is represented by its *resourceNumber* that is used to identify it. If no resource set is found, the client will get an empty response. On receiving a resource set, the client will process it. For each linked resource in it, the protocol *getResource* is executed. On this way, the resource will only be requested from the server, if it is not already cached. The *priortiyLevel* parameter depends on the kind of context element the resource set corresponds to. E.g if the resource set corresponds to location, the resource gets the *priorityLevel* from location. When a resource is still cached, but gets a member of a resource set that is from a context element with higher *priorityLevel*, the *priortiyLevel*

is set to the higher one. On the same way it can happen that the value of a context element gets empty (undefined situation). In this case the *priorityLevel* has to be adapted too. On that way context-aware selection is also realized by a combination of *requesting a resource set* and the *getResource* protocol. The main difference to traditional selection is that *getResource* is not triggered by a user interaction. It is triggered from the situation that occurs before the user explicitly request a model or the specific parts of it. The resources are hoarded for later usage. On this way preemptive caching is realized.

5.3.3 Cache Replacement Policy

The gain based function with priority as parameter (GBFP) is used for the replacement policy. In the prototype it is realized with the help of the priority list. With the list it is possible to determine, which resource in the cache will be replaced (or not). The policy is always executed, before inserting a resource in the cache. It is checked, if the resource can be cached, or not due to insufficient cache space and a low priority. How the policy is realized, can be seen in figure 5.9. As the policy is only executed on the client and has no interaction with the server, a flowchart is used.

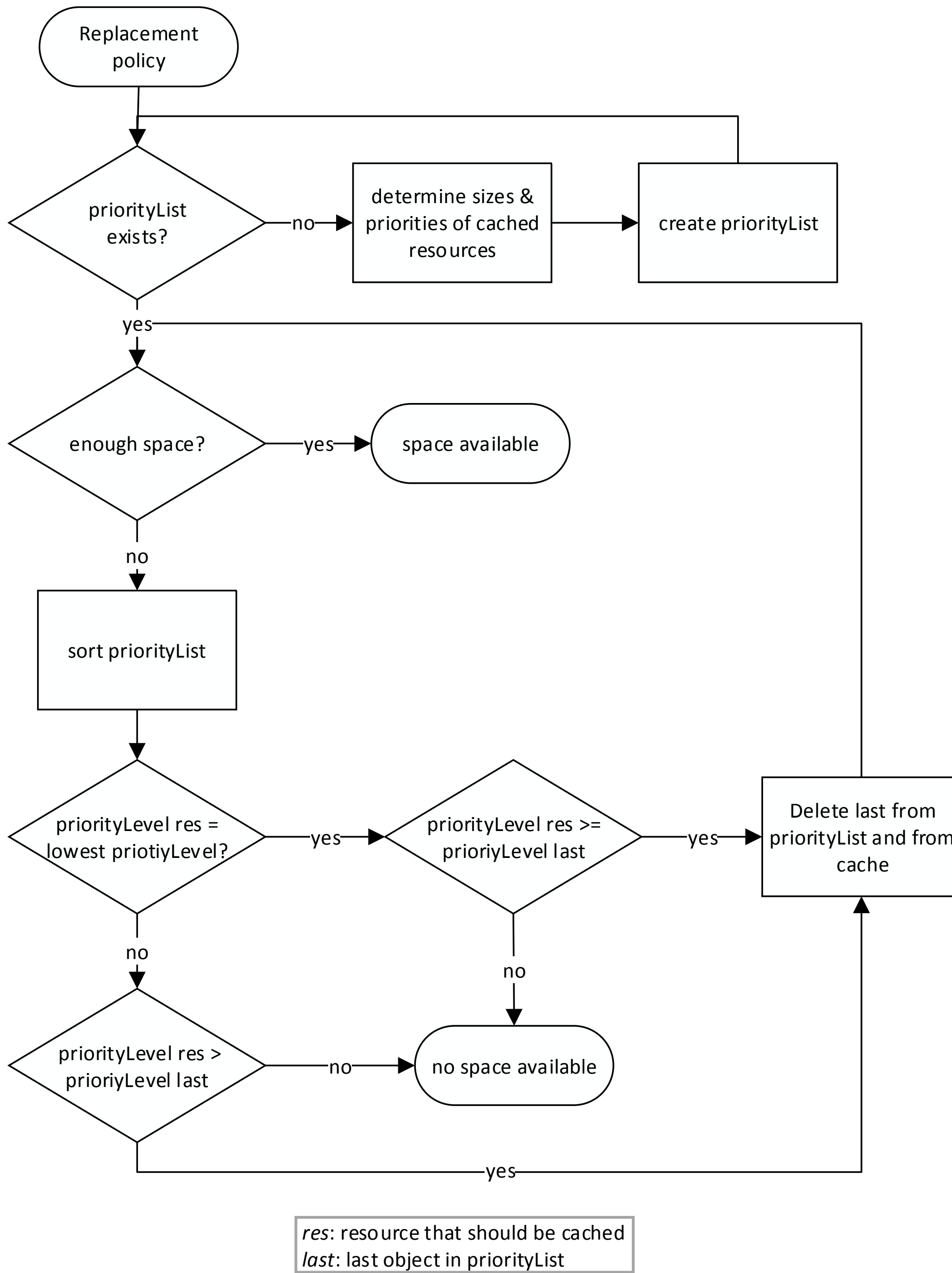

Figure 5.9: Replacement policy

When the policy is executed, it is checked if a *priority list* already exists. If not, a new one is created. Therefore, the identifier of all the resources in the cache are used to create objects that are added to list. During creating an object, the *size and priority of the resource is determined*. These are added to the object. For the determination of the priorities, the active situations are considered. If there is a resource set from an active situation on the client, all its corresponding resources get the *priority level* from the situation. If not the *priority level* is set to the one from traditional selection. When the *priority list* exists, it is checked if there is enough space in the cache, to store the resource. To calculate the free space, the first entry in the *priority list* is used. It includes the total size of the all the resources in the cache. If there is enough, the policy finishes with returning *space available*. If not, the *priority list is sorted* descending after the priority. Here also the access frequency is considered. A high access frequency can lead to a higher priority. On this way the resource with the lowest priority is located on the bottom of the list.

Now it depends on the *priority level* what is executed next. It can be extracted out of the priority. The first two numbers of the priority represent the *priority level*. (e.g., priority '20124' has priority level '20'). If the level from the resource that should be cached equals to the lowest ('10'), it is checked if its level is equal or greater than the level from the last object. There is no lower *priority level*. So when checking only for *greater than* (like it is done for all other *priority levels*), once the cache is full, no other resource with the lowest *level* can be inserted any more. If the last object in the list is suitable (has lower or same priority level) the resource identifier from the *priority list* is used to delete the resource from the cache. After that the object form the list is deleted too, and it is checked again, if there is enough space. If the last object is not suitable (has a higher priority level) the policy will return *no space available*. With this implementation of the replacement policy it is possible to manage the cache and protect specific resources for replacement. This depends on strategy that is used to select the resources that should be cached.

5.3.4 Cache Coherence Strategy

The Stateful Invalidation Protocol (SIP) is used for the coherence strategy. It is implemented over a WebSocket connection that is used to send the invalidation reports that includes the obsolete resources. This kind of connection is bidirectional that means the client and the server can send messages. When the server recognize that a client has obsolete resources in its cache, it sends the invalidation report to the client. To make this possible the server store all the resources that are requested by the client in the UserResource table. The server also knows the clients that are connected

over the WebSocket. On this way the invalidation reports are sent selective. Before this can be processed, the client has to initialize at the server. Figure 5.10 shows this initialization. The interaction between the client and the server is shown in a sequence diagram.

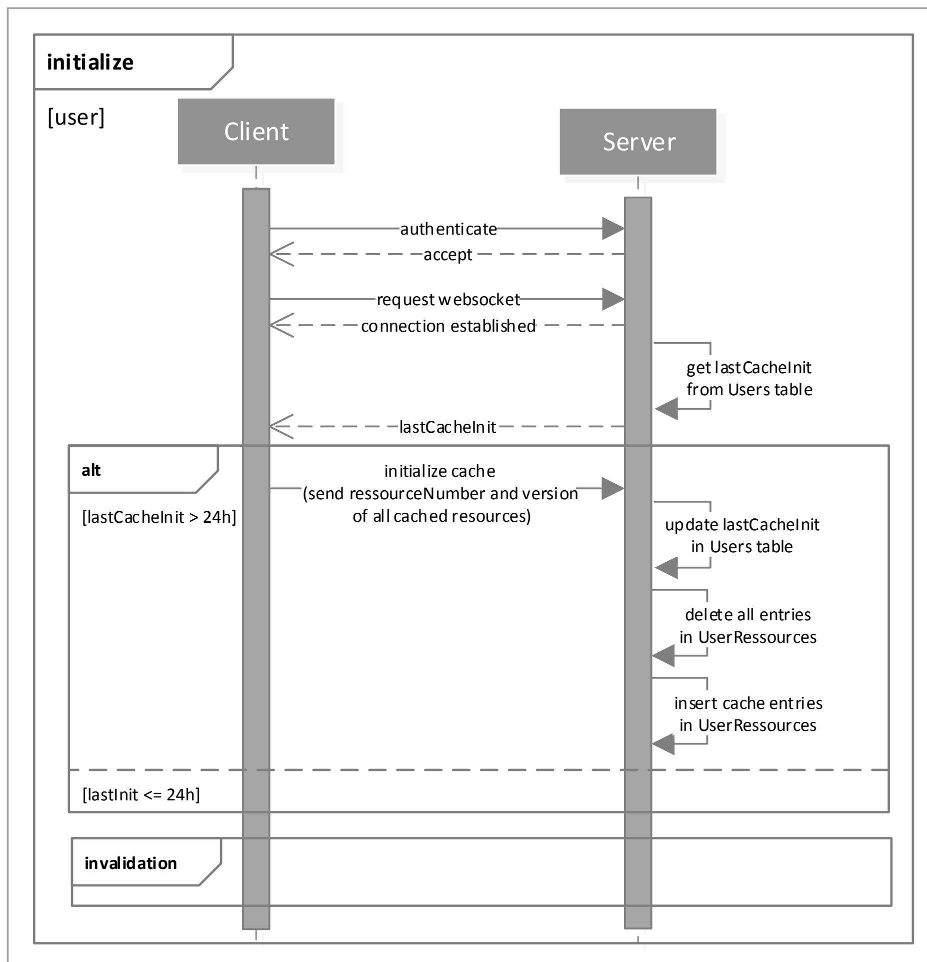

Figure 5.10: Initialization at the server

To create a *WebSocket* connection the client has to authenticate with the user credentials at the server. After establishing the connection, the server searches for the *lastCacheInit* of the user in the Users table of the database. This is the time when the cache of the

client was last initialized at the server. In the prototype the time window 'w' after that a client have to *initialize* again is set to 24 hours. So when the *lastCacheInit* is more than 24 hours ago, the client has to *initialize its cache*. Therefore, the client has to send all *resource numbers and its corresponding versions from the cache*, to the server. When the server receives that, the *lastCacheInit is updated* to the current date and time and all *entries in the UserResource table that corresponds to the user are deleted*. After that, the sent *resource numbers and versions are inserted in the table*.

When the *initialization* is done, the *invalidation* is started. This process is excluded to an extra diagram. It is presented in figure 5.11.

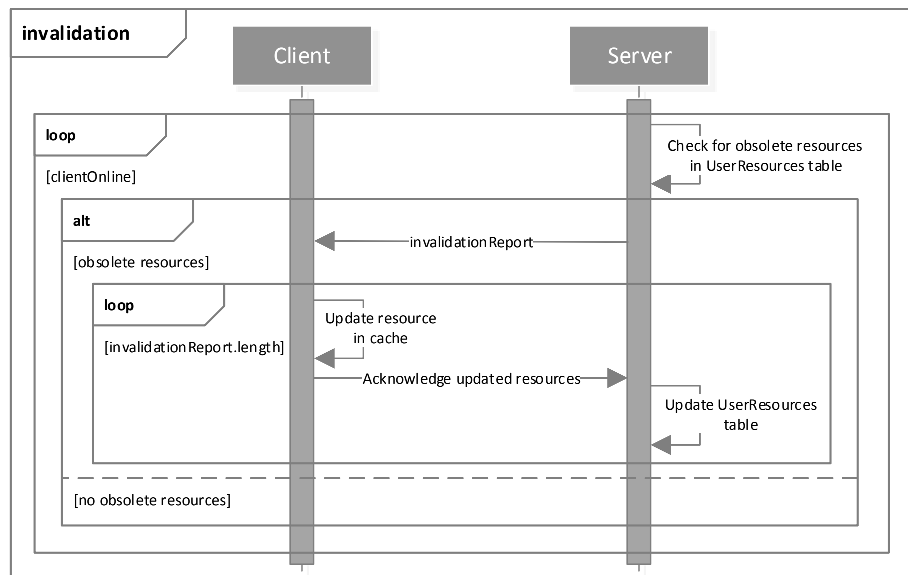

Figure 5.11: Protocol - invalidation

The invalidation protocol for a client is executed as long as the *client is online*. With the help of the version within the UserResource table *obsolete resources can be identified*. If a server can find some, an *invalidation report* that includes the resource number is sent to the client. On receiving, the clients *iterate through all resources in the report*. If the resource can be found in the cache, it gets deleted and the protocol *getResource* is used to get the new version. Thereby, the UserResource table is also updated. The

server gets informed that the client has the new version. If none resource is found (means that the resource was replaced), the client *acknowledge the receiving* with the information that the resource is not cached.

6 Evaluation

In this chapter, the results of this thesis are evaluated. The created caching concept is completely realized in the implemented prototype. Therefore, it is used for the evaluation of the concept. The only exception represents the dynamic context-aware selection strategy. This part is not implemented, as it is not necessary for the evaluation of the concept. It is enough to use static context-aware caching for the evaluation of preemptive caching (hoarding).

To give meaningful results, the evaluation was proceeded in a use case scenario of the app. The app is used to view different 3D models of the products within a web browser. Thereby, different actions were executed in the same order within the old app (the new architecture is not included) and the new app (includes the new architecture). Within the new app the steps were once executed with traditional selection only and once with location based selection. When using location based selection, the traditional selection runs also at the same time. Location based selection was chosen as the representative of the context-aware selection strategies. All the strategies perform on the same way. The difference is only the situation that comes from different contexts.

In the use case scenario an existing model was used, to perform the measurements. In figure 6.1 this model is presented by means of its logical data model. This contains the different branches of the product. Thereby it is marked, which resources are included in the resource set of the station where the app was used.

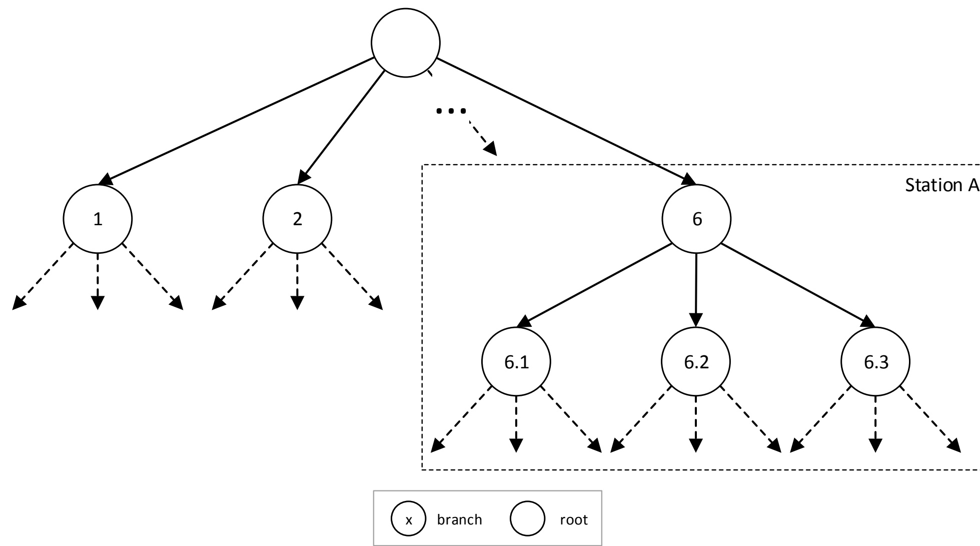

Figure 6.1: Use case for the evaluation

The product model consisted of *6 main branches*. Each of the branches had a different amount of nodes. Also the size differed. *Station A* was the location. At this station, the components of branch 6 were needed. This branch is divided into the branches *6.1, 6.2 and 6.3*. Therefore all the resources of these branches were included in the resource set of *station A*. For the measurements the branches 1, 2, 6.1, 6.2 and 6.3 were considered. Thereby, branch 1 and 2 were not part of the resource set of *station A*. This means, when using location based selection, three of five branches are cached preemptive.

The different actions that were executed within the app were *viewing a branch* and closing the app. A total amount of 10 actions were executed successively in each mode of the app (old app, new app with traditional, new app with location based). The actions from the different steps were chosen with an random function. The first step represented an exception. This step was always necessary as it loaded the structure resource from the back-end. In each of the app mode the same order was used. This was repeated for three times to get a different order of the actions. This leaded to a different behavior of the different modes. During the execution of the different actions, the time for loading the resources from the back-end was measured. Also the size of the transferred resources were recorded. At the end of the measurements the total waiting time and the total size of the transferred resources were was computed. An assumption that was made is that there was enough space in the cache on the device.

Also the situations are recognized early enough, so that the resources are still hoarded when they are needed. Furthermore, the connection was stable and no connection abort occurred.

To conduct the measurements, a machine running the Windows 8 operating system with 8 GB of RAM and an Intel(R) Core(TM) i5-4300U CPU @ 1,90GHz 2,49GHz processor was used. The app and the back-end were executed in two different processes. The app was deployed on an Apache web server and executed within a Google Chrome web browser. In this web browser all the HTML5 Offline Storage's (included Application Cache and IndexedDB) were supported. The back-end was deployed and executed within a Tomcat application server. To simulate the same speed for all app modes and measurements, the the DevTools[1] of Google Chrome were used. The speed was throttled to 30MB/s (equates a WiFi connection within Chrome). This tool was also used to measure the amount of transferred Megabyte over the network (size in MB) and the loading times of the resources (time in seconds). The measurement of scenario 1 is presented in Table 6.1:

[1] https://developer.chrome.com/devtools/docs/network

Speed: 30MB/s		**Old app**	**New app (location based)**	**New app (traditional)**
Step	Action	Time in s / (Size in MB)	Time in s / (Size in MB)	Time in s / (Size in MB)
0)	*Before first action (not considered in total waiting time)*	*0 (0)*	5,7 (4,62)	*0 (0)*
1)	Get structure	0 (0)	0,1 (0,03)	0,1 (0,03)
2)	View branch 6.1	8,2 (17,7)	0,15 (0)	2,4 (1,9)
3)	View branch 6.2	0 (0)	0,2 (0)	2,8 (2,3)
4)	Close app	-	-	-
5)	View branch 6.3	8,2 (17,7)	0,05 (0)	0,5 (0,42)
6)	View branch 6.1	0 (0)	0,15 (0)	0,15 (0)
7)	View branch 6.3	0 (0)	0,05 (0)	0,05 (0)
8)	View branch 1	0 (0)	0,2 (0)	0,2 (0)
9)	View branch 6.2	0 (0)	0,15 (0)	0,15 (0)
10)	View branch 6.1	0 (0)	0,15 (0)	0,15 (0)
Total waiting time in s / (Total size in MB)		**16,4 (35,4)**	**3,65 (9,75)**	**8,95 (9,75)**

Table 6.1: Measurement of scenario 3

Step 0 is the time, before the first action was performed. During this time, the new app with location based selection got active. The resources from branches 6.1, 6.2 and 6.3 that were included in the resource set from station A were requested and cached. The time that is needed for this step was not considered in the total waiting time. This did not affect the effective waiting time when performing the different actions. The first action that was performed was *get structure*. In the old app this was not executed. The complete product model (structure and components) was requested from the back-end when performing the first action. There is no option to request only specific parts of the model. In the location based and traditional app mode, this step was necessary.

The action from step 2 is executed in each app mode. In the old app, the complete model is requested. Within the location based app mode the resources were still cached. Therefore, they were loaded from the cache. As in the traditional app mode these resources are loaded for the first time, they have to be requested from the back-end. In step 3 the old app mode had the complete model from step 2. There was no need to load branch 6.2. The location based app mode got the resource from the cache and

the traditional app mode had to request the resources for the first time from the server again. At step 4 the app was closed in all modes. When it woke up at step 5, in the old app the whole model had to be requested again from the back-end. All resources got lost. In the new app, both modes were not influenced through the closing. The already cached resources were still in the cache. This went further until the scenario ended with step 10.

The results of this measurement showed that the location based selection performs best in this scenario. It was possible to achieve the best cache hit ratio that results in a short waiting time and a reduced amount of transferred MB over the network. As no resources were hoarded in in the traditional app mode, the hit ratio was lower. However, this performed better than the old app.

To confirm the results, two further scenarios were created. The actions of the steps were also randomly chosen. In Table 6.2 the results of scenario 2 are presented. Table 6.3 shows the results of scenario 3.

Speed: 30MB/s		**Old app**	**New app (location based)**	**New app (traditional)**
Step	Action	Time in s / (Size in MB)	Time in s / (Size in MB)	Time in s / (Size in MB)
0)	*Before first view (not considered in total waiting time)*	*0 (0)*	5,7 (4,62)	*0 (0)*
1)	Get structure	0 (0)	0,1 (0,03)	0,1 (0,03)
2)	View branch 2	8,2 (17,7)	1,95 (3)	1,95 (3)
3)	Close app	-	-	-
4)	View branch 1	8,2 (17,7)	2,6 (5,1)	2,6 (5,1)
5)	View branch 6.1	0 (0)	0,15 (0)	1,9 (2,4)
6)	View branch 1	0 (0)	0,35 (0)	0,35 (0)
7)	Close app	-	-	-
8)	View branch 1	8,2 (17,7)	0,35 (0)	0,35 (0)
9)	View branch 6.3	0 (0)	0,05 (0)	0,5 (0,42)
10)	View branch 6.3	0 (0)	0,05 (0)	0,05 (0)
Total waiting time in s / (Total size in MB)		**24,6 (53,1)**	**5,55 (12,72)**	**8,3 (10,45)**

Table 6.2: Measurement of scenario 2

In this scenario the app was closed two times. This leaded to a bad result for the old app. The model had to be requested again for three times. The app that used location based selection requested more resources than needed. In the result set the resources from branch 6.2 are included, but not viewed. In this case the app using traditional selection requested less resources. However, with regard to time, location based selection performed better.

Speed: 30MB/s		**Old app**	**New app (location based)**	**New app (traditional)**
Step	Action	Time in s / (Size in MB)	Time in s / (Size in MB)	Time in s / (Size in MB)
0)	*Before first view (not considered in total waiting time)*	*0 (0)*	*5,7 (4,62)*	*0 (0)*
1)	Get structure	0 (0)	0,1 (0,03)	0,1 (0,03)
2)	View branch 2	8,2 (17,7)	1,95 (3)	1,95 (3)
3)	View branch 3.3	0 (0)	0,05(0)	0,5(0,42)
4)	View branch 2	0 (0)	0,25 (0)	0,25 (0)
5)	View branch 1	0 (0)	2,6 (5,1)	2,6 (5,1)
6)	View branch 3.2	0 (0)	0,2 (0)	2,8 (2,3)
7)	View branch 3.3	0 (0)	0,05 (0)	0,05 (0)
8)	View branch 3.3	0 (0)	0,05 (0)	0,05 (0)
9)	View branch 2	0 (0)	0,25 (0)	0,25 (0)
10)	View branch 3.1	0 (0)	0,15 (0)	2,4 (1,9)
Total waiting time in s / (Total size in MB)		**8,2 (17,7)**	**5,65 (12,75)**	**10,95 (12,75)**

Table 6.3: Measurement of scenario 3

In this scenario, the app was never closed. However, the app using location based selection performed better anyway. The traditional selection instead, needed more time for loading all the resources. This means, when working with the same model over a longer time without closing the app the old app performs better. But this seems to be an unrealistic scenario.

These results of the measurement are used to evaluate the single key features of the caching concept:

- **Reduced waiting time by use of caching.**
  In all scenarios location based caching performed best with regard to the waiting times. Through the combination of preemptive caching through location based selection (context-aware selection) and traditional caching more cache hits are achieved. This results in a reduced waiting time. Also traditional caching achieved in two of three cases a reduced waiting time compared with the old app that do not cache anything.

- **Fewer network transmissions by use of caching.**
  The size of the transferred resources were reduced in all scenarios by caching. Once cached, the resources can be accessed from the cache. The app spent less time in requesting and receiving resources. That saves a lot of energy.

- **Robustness against intermittent connectivity by use of preemptive caching.**
  To simulate an abort of the connection the DevTools from Google Chrome were used. In scenario 6.1 the connection aborted at *step 2*. After the disconnect the loading of the model failed in the old app. With the usage of location based selection it was still possible to view the branch. As all the resources were cached before viewing the branches at the station, the aborting connection did not affect the loading of the branch.

- **Usage of hoarding to enable an offline mode.** Loading the old app without a connection was not possible. A connection was needed to start the app. Through the Application Cache of the new architecture, it was possible to start the app without a connection. All the core files of the app were stored on the device. Also the components of branches 6.1, 6.2 and 6.3 could be viewed, as they were stored in the cache before the devices went offline. On this way hoarding enables an offline mode.

7 Conclusion and future prospects

In this master thesis a concept for a caching mechanism within a mobile app in the engineering domain is presented. The concept is designed for a concrete app that allows to view 3D models of products in a web app. However, the concept can also be used for other apps. With this concept, it is possible to compensate the technology oriented problems from apps in the engineering domain. It leads to a reduced waiting time, the amount of transmission are reduced and the problems with intermittent connectivity are compensated. Also an offline mode is provided by the concept.

In the thesis, different strategies are considered that can be used for a caching mechanism. These strategies are grouped into four characterizations. Cache Granularity, Cache Selection Strategy, Cache Replacement Policy and Cache Coherence Strategy. Within the concept, for each characteristic, one strategy comes to use. The cached resources are Objects (Cache Granularity). With a gain based function that uses priorities (GFBP) the replaced resources are determined (Cache Replacement Policy). A stateful invalidation protocol (SIP) is used to invalidate obsolete resources in the cache of the clients (Cache Coherence Strategy). The Selection Strategy constitutes an exception. Thereby, multiple strategies are used. To select the resources that have to be cached, traditional and preemptive selection strategies are used. Within the preemptive approach, context-aware selection strategies come to use. On the basis of context, the resources that are needed in the current situations are determined and cached. In contrast to traditional selection, the resources are provided to the app without an explicit request. Hence, when the resources are needed, a fast loading directly from the device is possible. Through this hoarding of resources, it is possible to use the app offline.

With a concrete implementation of the concept, it was possible to evaluate the results. For that purpose, an existing app was used and adapted. The single steps of the implementation are described by detail in this thesis. It is described, how the architecture is changed and which protocols and functions are needed to realize the concept. Through the evaluation it was possible to confirm all the provided key features by the concept.

In the large, the caching concept can be used in overall solutions for mobile apps in the engineering domain that use context-awareness to enable smart access to product data. Hoos et al. create an architecture for mobile context-aware product data management (MoCa-PDM) that make such a smart access possible [HSM16]. Thereby caching and hoarding plays an important role. Together with context-aware data provisioning that is supported by the concept, they build the data management components of the architecture [HSM16].

As future work, the realization of the recognition of the situations within the concept should take place. The thesis engages only with the providing of the resources by means of the situations. How the values of the context elements come about is not considered. Real sensors of mobile devices like GPS or the camera should be used to get the different values.

Bibliography

[ADB+99] G. D. Abowd, A. K. Dey, P. J. Brown, N. Davies, M. Smith, and P. Steggles. "Towards a better understanding of context and context-awareness." In: *Handheld and ubiquitous computing*. Springer, 1999, pp. 304–307 (cit. on pp. 10, 18, 24).

[AKL] D. S. Azevedo, G. Karaman, and D. Lehmann. "Concept of Mobile Product Data Interaction." In: () (cit. on pp. 45, 48).

[Aro08] J. Arokiamary. *Mobile Computing*. Technical Publications Pune, 2008 (cit. on pp. 19, 20).

[BBFS11] M. Berekovic, U. Brinkschulte, W. Fornaciari, and C. Silvano. *Architecture of Computing Systems - ARCS 2011*. Springer, 2011 (cit. on pp. 9, 14).

[BI94] D. Barbará and T. Imieliński. "Sleepers and workaholics: caching strategies in mobile environments." In: *ACM Sigmod Record*. Vol. 23. 2. ACM. 1994, pp. 1–12 (cit. on p. 20).

[Com05] P. Components. *UGS, Open product lifecycle data sharing using XML, Write paper: PLM XML*. 2005 (cit. on p. 47).

[DBM+07] L. Ding, A. Ball, J. Matthews, C. A. McMahon, and M. Patel. "Product representation in lightweight formats for product lifecycle management (PLM)." In: *4th International Conference on Digital Enterprise Technology* (2007) (cit. on p. 47).

[FCAB98] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. "Summary cache: A scalable wide-area web cache sharing protocol." In: *ACM SIGCOMM Computer Communication Review*. Vol. 28. 4. ACM. 1998, pp. 254–265 (cit. on p. 14).

[FZJF06] W. Feng, L. Zhang, B. Jin, and Z. Fan. "Context-aware caching for wireless internet applications." In: *e-Business Engineering, 2006. ICEBE'06. IEEE International Conference on*. IEEE. 2006, pp. 450–455 (cit. on pp. 14, 18, 25).

[GAAU15] Y. Go, N. Agrawal, A. Aranya, and C. Ungureanu. "Reliable, consistent, and efficient data sync for mobile apps." In: *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (2015) (cit. on p. 27).

[Gol03] C. Gollmick. "Replication in mobile database environments: a client-oriented approach." In: *null*. IEEE. 2003, p. 980 (cit. on p. 28).

[HGM15] E. Hoos, C. Gröger, and B. Mitschang. "Mobile apps in engineering: a process-driven analysis of business potentials and technical challenges." In: *Procedia CIRP* 33 (2015), pp. 17–22 (cit. on p. 9).

[HSM16] E. Hoos, C. Stach, and B. Mitschang. "Mobile Context-Aware Product Data Management (MoCa-PDM): Towards Smart Processing of Big Engineering Data on Mobile Devices." In: (2016) (cit. on pp. 9, 10, 21, 22, 24, 25, 32, 33, 80).

[HWM09] H. Höpfner, S. Wendland, and E. Mansour. "Data caching on mobile devices." In: *ICSOFT 2009-4th International Conference on Software and Data Technologies*. 2009 (cit. on pp. 15, 28).

[JEHA97] J. Jing, A. Elmagarmid, A. S. Helal, and R. Alonso. "Bit-sequences: an adaptive cache invalidation method in mobile client/server environments." In: *Mobile Networks and applications* 2.2 (1997), pp. 115–127 (cit. on p. 20).

[JJ99] J. Jing and A. Joshi. *Mobile Data Management and Applications*. Springer Science & Business Media, 1999 (cit. on pp. 18, 19).

[KH08] A. van Kesteren and I. Hickson. *Offline Web Applications*. Tech. rep. W3C, 2008. URL: http://www.w3.org/TR/offline-webapps/ (cit. on p. 51).

[KR01] U. Kubach and K. Rothermel. "Exploiting location information for infostation-based hoarding." In: *Proceedings of the 7th annual international conference on Mobile computing and networking*. ACM. 2001, pp. 15–27 (cit. on p. 15).

[LS97] H. V. Leong and A. Si. "On adaptive caching in mobile databases." In: *Proceedings of the 1997 ACM symposium on Applied computing*. ACM. 1997, pp. 302–309 (cit. on pp. 18, 19).

[MSG+15] N. Mehta, J. Sicking, E. Graff, A. Popescu, J. Orlow, and J. Bell. *Indexed Database API*. Tech. rep. W3C, 2015. URL: https://www.w3.org/TR/IndexedDB/ (cit. on p. 51).

[PBHS11] M. Peters, C. Brink, M. Hirsch, and S. Sachweh. "A client centric replication model for mobile environments based on RESTful resources." In: *Proceedings of the Workshop on Posters and Demos Track*. ACM. 2011, p. 22 (cit. on p. 27).

[PS12] E. Pitoura and G. Samaras. *Data management for mobile computing*. Vol. 10. Springer Science & Business Media, 2012 (cit. on p. 9).

[Rot15] K. Rothermel. “Mobile Computing.” In: *University of Stuttgart* (2015) (cit. on p. 9).

[RP07] R. Rathore and R. Prinja. “An Overview of Mobile Database Caching.” In: *CiteSeerX, doi* 10.1.100 (2007), p. 9481 (cit. on pp. 15–17, 20).

[SI08] A. Saaksvuori and A. Immonen. *Product lifecycle management*. Springer Science & Business Media, 2008 (cit. on pp. 11, 22).

[SMC+14] D. Sethia, S. Mehta, A. Chowdhary, K. Bhatt, and S. Bhatnagar. “MRDMS-mobile replicated database management synchronization.” In: *Signal Processing and Integrated Networks (SPIN), 2014 International Conference on*. IEEE. 2014, pp. 624–631 (cit. on p. 27).

[SSAS11] S. Sulaiman, S. M. Shamsuddin, A. Abraham, and S. Sulaiman. “Intelligent web caching using machine learning methods.” In: *Neural Network World* 21.5 (2011), p. 429 (cit. on p. 18).

[Tan07] D. Taniar. *Encyclopedia of mobile computing and commerce*. IGI Global, 2007 (cit. on p. 16).

[WPG14] C. Wenzelmann, C. Plass, and J. Gausemeier. *Zukunftsorientierte unternehmensgestaltung: Strategien, geschäftsprozesse und it-systeme für die produktion von morgen*. Carl Hanser Verlag GmbH Co KG, 2014 (cit. on pp. 12, 13).

[XHLL04] J. Xu, Q. Hu, W.-C. Lee, and D. L. Lee. “Performance evaluation of an optimal cache replacement policy for wireless data dissemination.” In: *Knowledge and Data Engineering, IEEE Transactions on* 16.1 (2004), pp. 125–139 (cit. on pp. 18, 19).

All links were last followed on April 01, 2016.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature