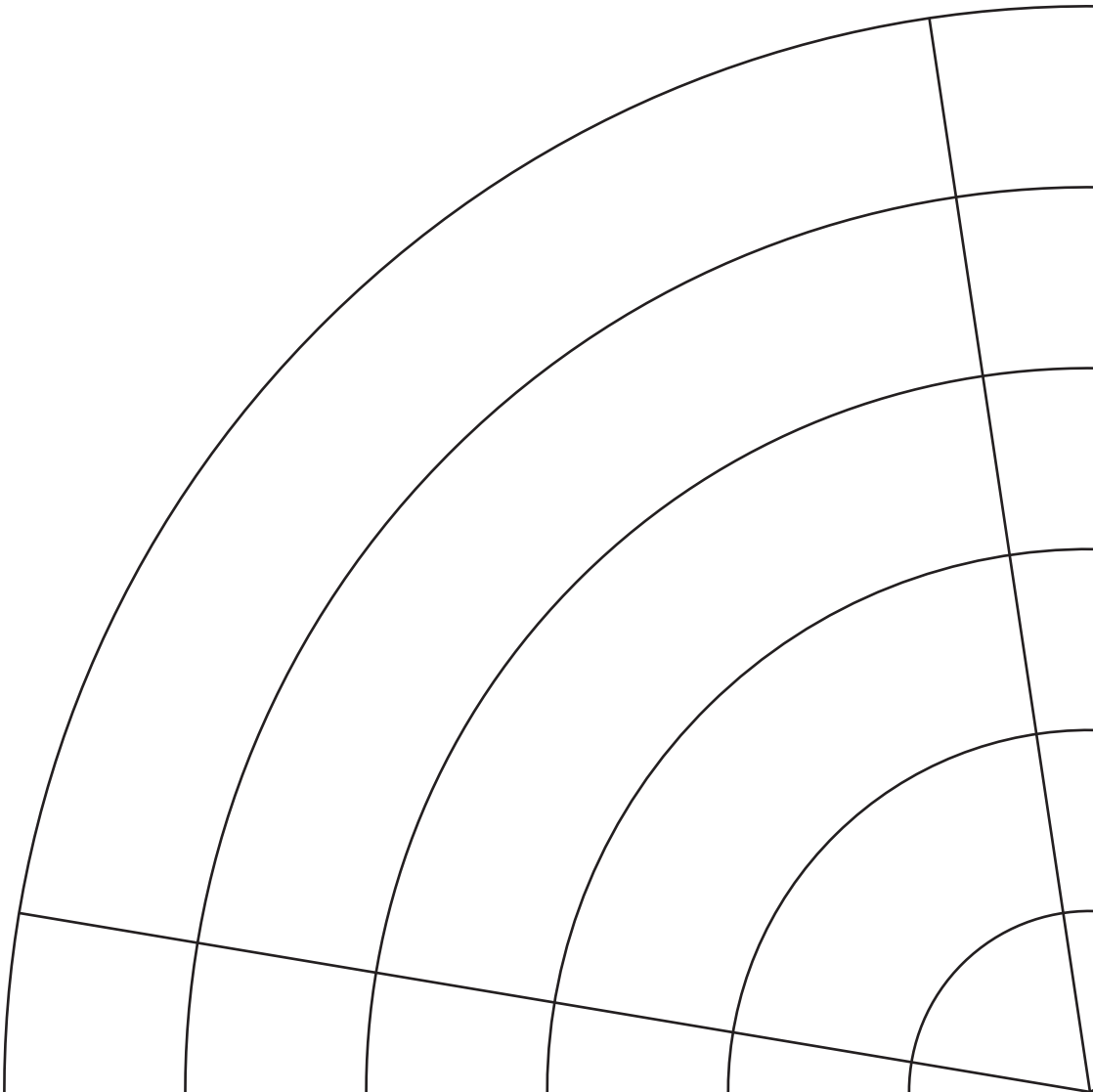
Travlendar+

Design Document

Barcella Lorenzo - Bellini Alberto Mario - Cavalli Luca



Travlendar+ DD

Table of Contents

1. INTRODUCTION

1.A Purpose

The purpose of this Design Document is to provide more technical information than the RASD by the representation of the architecture, design choices and interfaces made for Travlendar+ system. This document also includes the description of the components of the system and their interaction.

This document is addressed to the developers.

1.B Scope

Travlendar+ is an application designed to support movements in and across cities. The initial geographical scope will be a single city, but we mean to address a much wider area.

1.C Definitions, Acronyms, Abbreviations

1.C.1 Definitions

- **Task**: in the context of this document a task is considered to be the allocation of a precise amount of time in a given interval. It may optionally include the specification of a geographical place where the user should be at the start of

the task. What is done in the allotted time is not important for our purpose: it may be a meeting, an appointment, a break, a step in a tour, or anything else.
- **Appointment**: a task having a specified geographical place of attendance.
- **Basic mean of transportation**: one of the following: walking, personal biking, personal driving.
- **Fully covered area**: a geographical area for which we have data about all available and supported means of transportation (if more than one service of the same type is available in this area, we must have data of at least the major one).
- **Covered area**: a geographical area for which we have data about at least one more mean of transportation other than basic ones.
- **Partially covered area**: a geographical area for which we have complete information about all and only the basic means of transportation.
- **Uncovered area**: a geographical area for which we lack information about at least one of the basic means of transportation.
- **Combined path**: a mixture of transportation means used to reach one location from another one.
- **Best Mobility Option**: either a combined or a simple path that satisfies a set of constraints and maximizes preference options.
- **Alternative Mobility Option**: either a combined or a simple path that satisfies a subset of user constraints and maximizes preference options. Proposed only when no Best Mobility Option can be found.
- **Time Coverage**: a time parameter that defaults to 24 hours. Time Coverage puts a limit to the depth of the mobility solutions provided by the application: tasks over the Time Coverage in the future will not be considered for travel calculations. TC is considered a general preference and thus it is user definable, but it cannot exceed 120 hours.
- **Maximum/Minimum Destination Waiting Time**: the maximum/minimum time a valid schedule is allowed/requested to consider waiting at the destination location when computing for a Best Mobility Option. These user-definable options default to 10/5 minutes.

1.C.2 Acronyms & Abbreviation

- **T+** : Travlendar+
- **BMO**: Best Mobility Option
- **AMO** : Alternative Mobility Option
- **TC**: Time Coverage
- **MDWT**: Maximum Destination Waiting Time
- **mDWT**: Minimum Destination Waiting Time
- **GPSG:** GPS gadget
- **T1** : Tier 1
- **T2** : Tier 2
- **T3** : Tier 3

1.D Revision history

1.E Document Structure

This document describes the architecture of the T+ system providing a general overview of its structure and details on critical points, as well as a link to the goal of the system itself. The structure of the document will follow this line:

- ARCHITECTURAL DESIGN:
  We provide a general overview of the architecture of the system, the layers it's divided into and the components it's constituted of. We also describe how these components are expected to interact in different scenarios.
- ALGORITHM DESIGN
  Here the algorithms used in performance-critical tasks are described to addess solutions to performance requirements

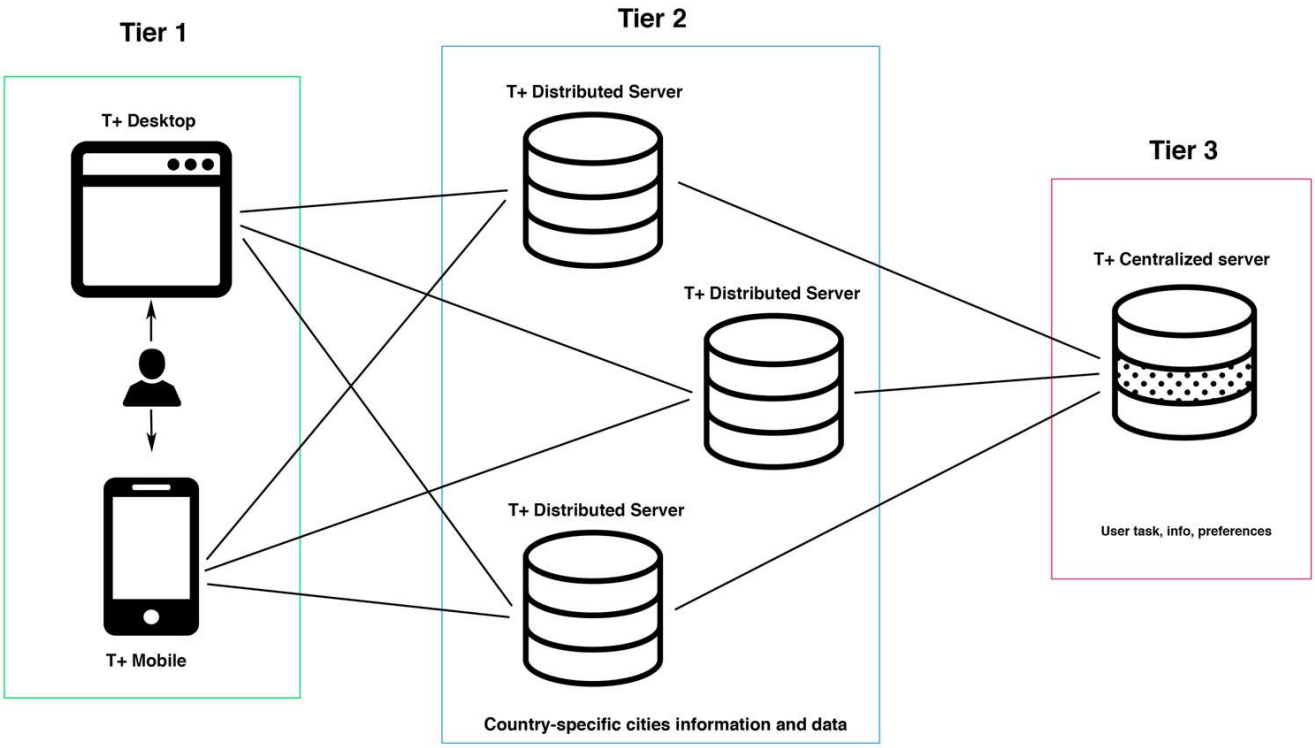
- USER INTERFACE DESIGN
  We describe how the user interface should be structured with UX diagrams
- REQUIREMENTS TRACEABILITY
  Goals of the T+ system are linked to the functionality of components described in the architectural design
- IMPLEMENTATION, INTEGRATION AND TEST PLAN
  Here a detailed plan on implementation and integration testing is provided in order to highlight what are the critical points and what are the dependencies in the system and the expected complexity of components

2. ARCHITECTURAL DESIGN

2.A Overview

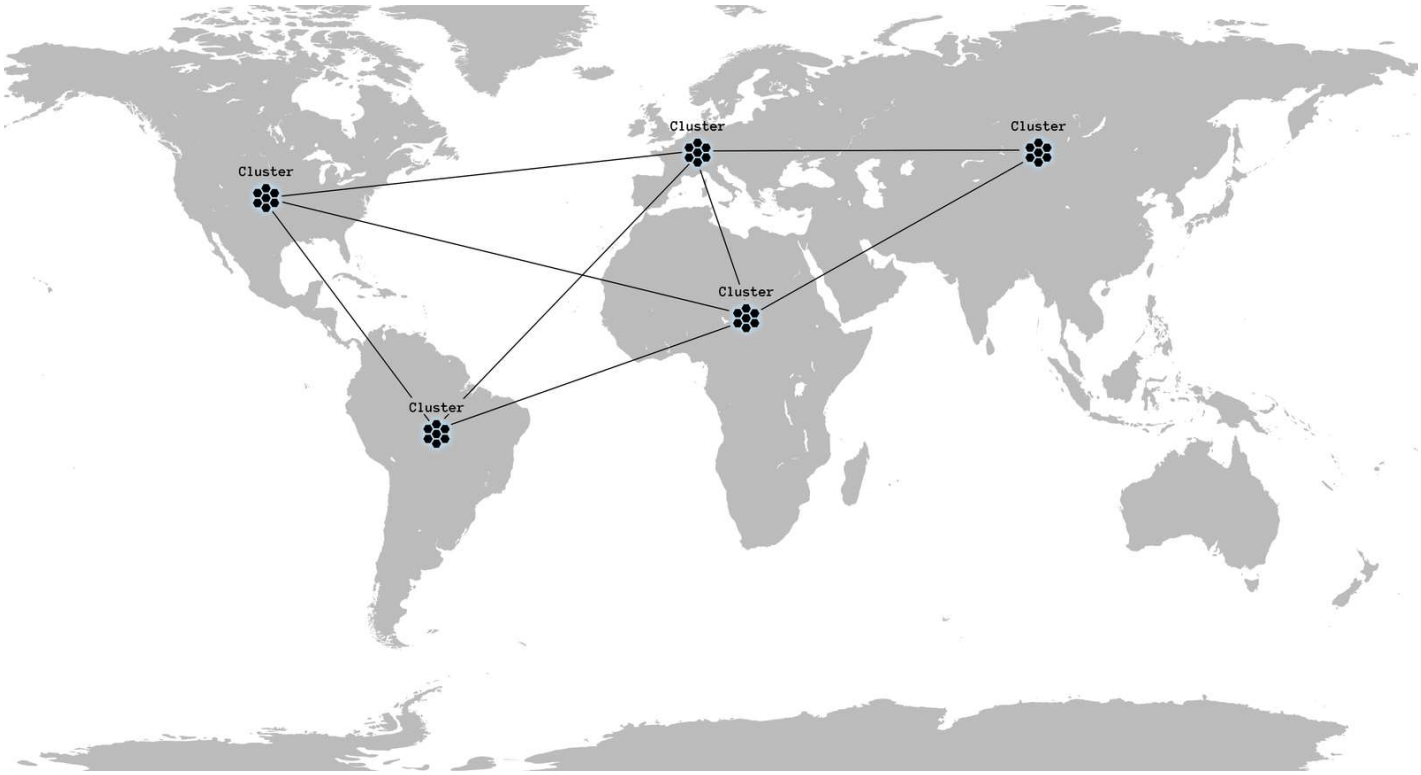
T+ architecture will follow a distributed system paradigm in which we will distinguish several different components: We are going to adopt a three tier architecture with a set of distributed servers that interact both with T+ Mobile or Desktop and some centralised servers to keep personal data secure.
Furthermore all architectural components are going to be clustered across the world while the system scales in coverage, as described later on.
Focusing on the tiers we will distinguish them as follows:

- **Tier 1** : T+ Mobile or Desktop that interact with tier 2.
- **Tier 2** : Distributed servers that interact with Tier 1 and 3.
- **Tier 3** : Centralised server that interacts with tier 2.

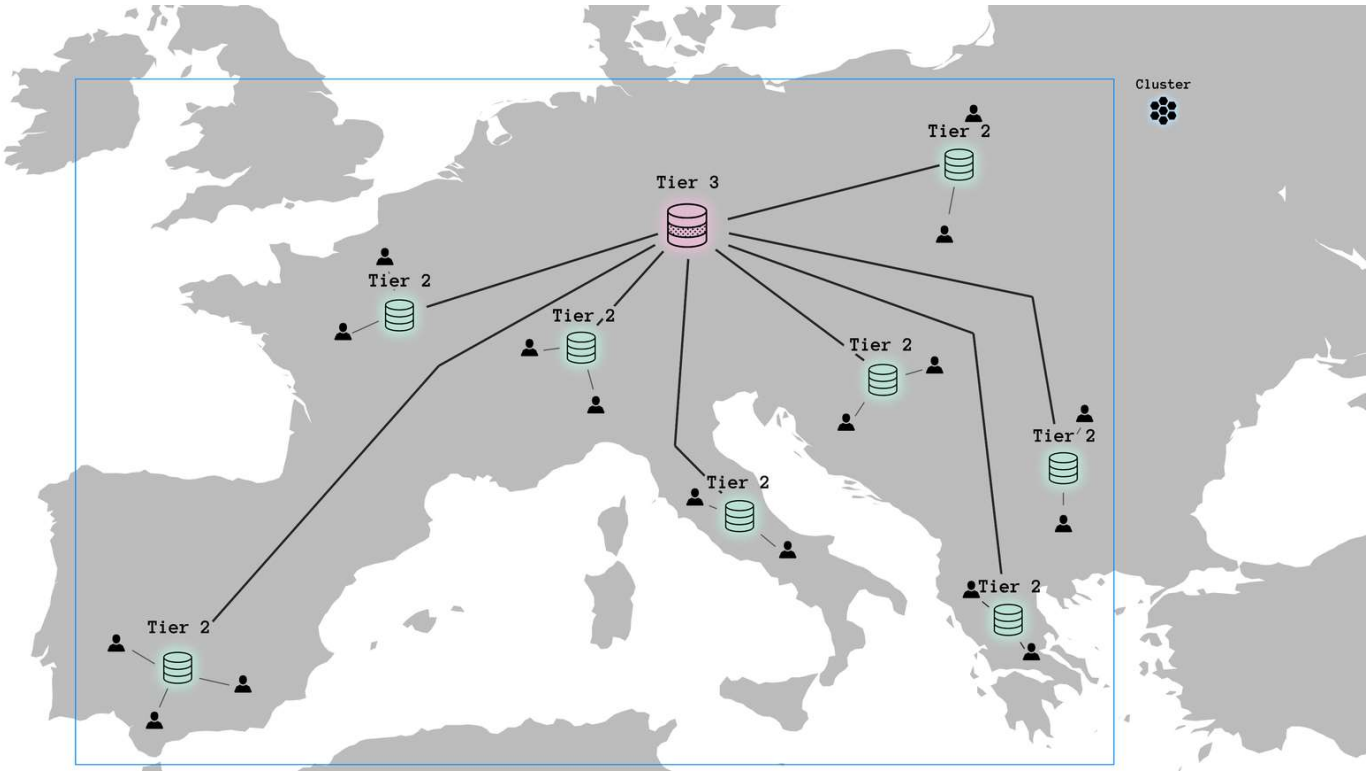


Three tier cluster

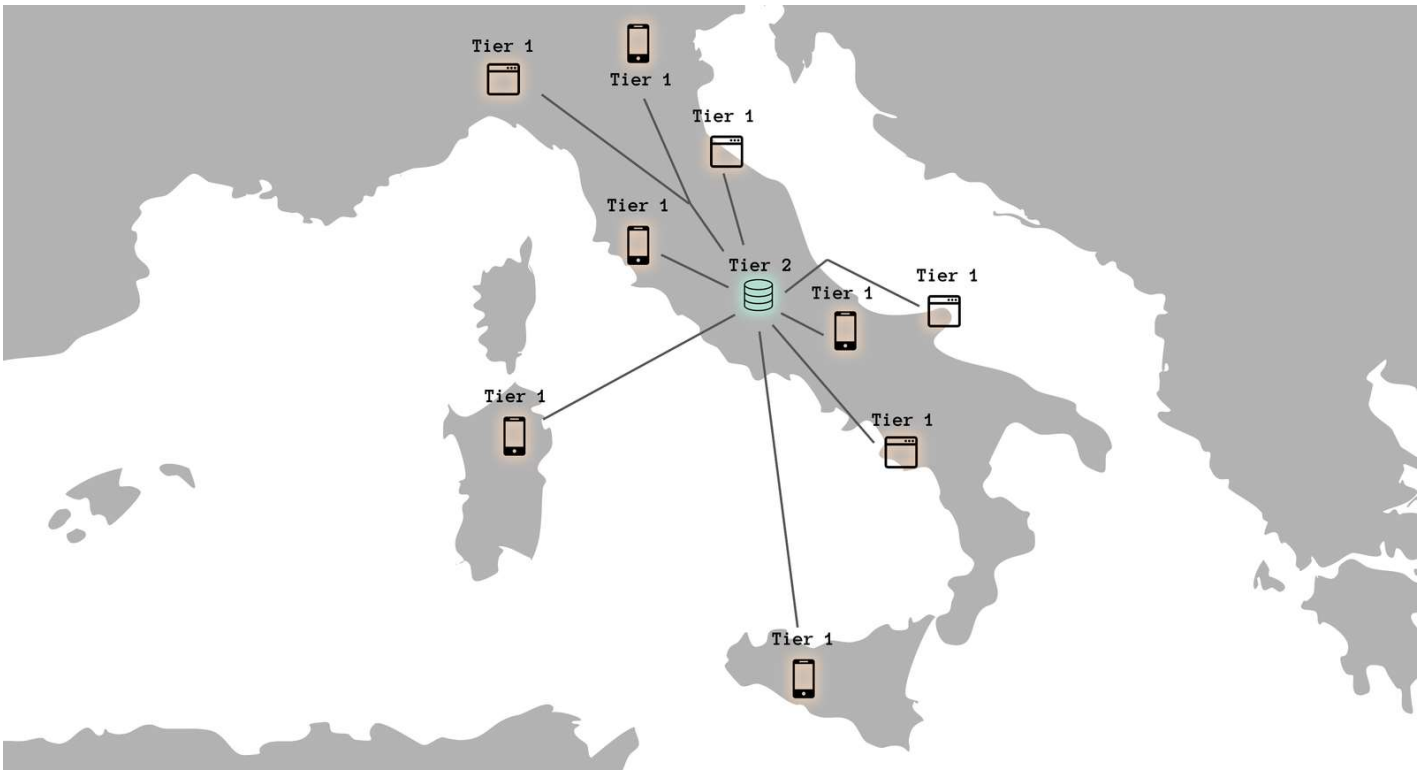
Our objective is to have a dedicated cluster for each continent we cover. Each cluster will consist of a tier 3 server and a scalable number of tier 2 servers, distributed across the countries of the continent, one for each country. Clearly tiers 2 and 3 will be present only in the countries and continents with any covered city, the system will grow gradually.



Clusterized architecture overview



Single cluster overview

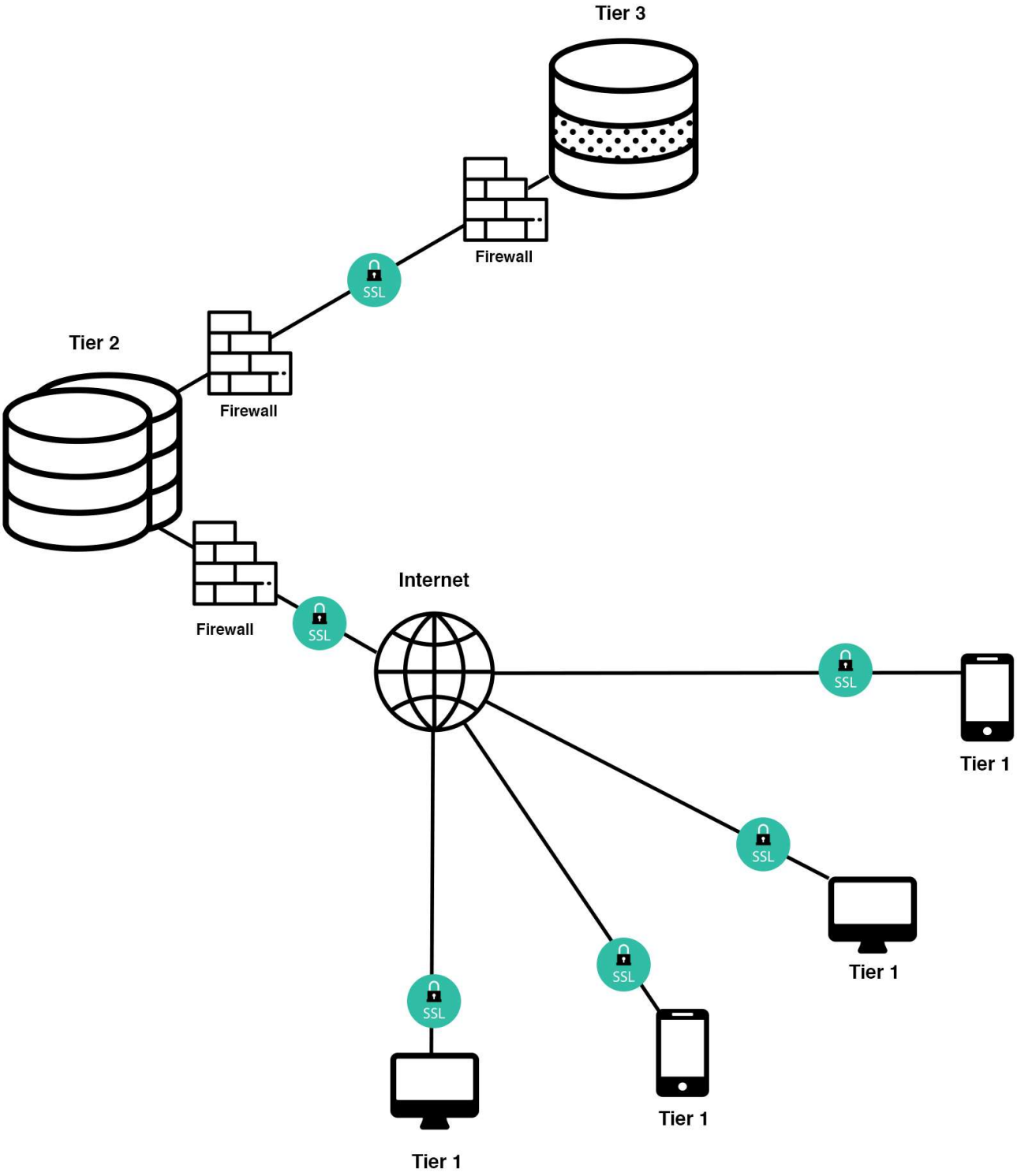


Single Tier 2 overview

On the one hand the distributed servers are going to host the main application and handle all the client requests.
The decentralised architecture focuses on maintaining locally all the information needed to be fully operative in the specific country: we don't want to replicate the same information on every Tier 2 server across the world. This allows us to parallelize most of the load of our servers, thus making the system very scalable.
The information held by each T2 server will mainly consist of travel mean datasets for the whole country and is going to be kept up to date automatically by the server itself: it is going to interact with local travel means APIs to fetch new data. Thus, the server in Italy, for instance, will interact with all the local APIs services in order to keep track of transportation mean schedule changes in Milan, Turin, Rome and all the cities covered by T+ in the same country.
If required T2 servers are even capable of computing a local BMO themselves and provide it to the clients that request it.
In order to keep the system balanced each user will make requests toward the T2 server located in the country from where he is using T+ and the T2 server is going to feed periodically the clients with travel mean updates (schedule changes, availability of new services etc.).
This architecture grants high scalability because it doesn't centralise all the information about the cities in single location but subdivides them into blocks that could singularly be managed.

On the other hand the Tier 3 server adopts a centralised architecture and is going to be highly isolated from the internet for security reasons: user information are personal and we don't want unauthorized people to access it.
T3 is going to be connected to T2 servers throughout a secure connection that makes use of Secure Socket Layers (SSL) and will interact only with them.
T3 servers are going to hold all the information about the users: task schedule, preferences, profile details and passes.
The centralised architecture helps T3 Servers to perform analysis on user behaviour and habits in order to constantly enhance our service.
Since T3 is not directly connected with T1, the second tier is going to forward user data back and forth to the clients to collect new information and synchronize them across all the different user devices.

To give a better understanding of the network infrastructure the following schema has been defined:



Network infrastructure

Finally the first tier is defined by the T+ Desktop & Mobile applications and the user is going to interact with these components directly.

T+ Mobile is going to be a fat client: as described in the RASD, the BMO computations are usually going to be done within the application itself (sometimes on Tier 2) and sporadically some requests will be made to the server to notify about new events that the user might generate (new task creation, personal information changes and so on).

The fat client will grant the reliability requirement for the core services, thus it will include:

- A **local database** component to give persistence to tasks, appointments, account data, payment data, available tickets, constraints and preferences
- A **transportation database** component with all locally downloaded transportation data
- A **transportation data management** component to provide the abstraction of a full graph while optimizing memory usage and transportation database access
- An **account manager** to provide a structured interface to user related data, manage tasks in the form of a calendar and manage all constraints and preferences
- An **optimizer** able to compute and update BMOs and lazily precompute heuristics on the available graph to speed up later computation (see section 3 about algorithm design)
- A **payment component** to manage payments to third parties directly in the client without the need to send payment data to the T+ servers

Other than that, the client must provide the classical functionalities with the following components:

- A **UI management** component.
- A **network interface** component to manage the server services

The T+ Desktop application is going to be a thin client due to the side functionality it will provide: no computation is going to be made here, the user will be only capable of creating new tasks, view his schedule and update his preferences. This application interacts with the server throught HTTP requests to fetch the data needed.
The Desktop architecture objective is to grant data management and synchronization across different T+ applications, either Desktop or Mobile. Even though it is not going to exhibit all the functionalities implemented in T+ Mobile, still some components will be required in order to manage these functionalities.
Some of these components recall the ones used in T+ Mobile and, with possible little changes, will be:

- A **UI management** component that is going to interact with the user directly.
- A **network interface** component to access and share data with the server.
- An **Account manager** component to handle all the user preferences, constraints and calendar tasks.

The application server architecture will have to grant the services of personal account management, device synchronization and transportation update feeds. To achieve this, some client low level components will be used into this architecture as well, with different high level back-end:

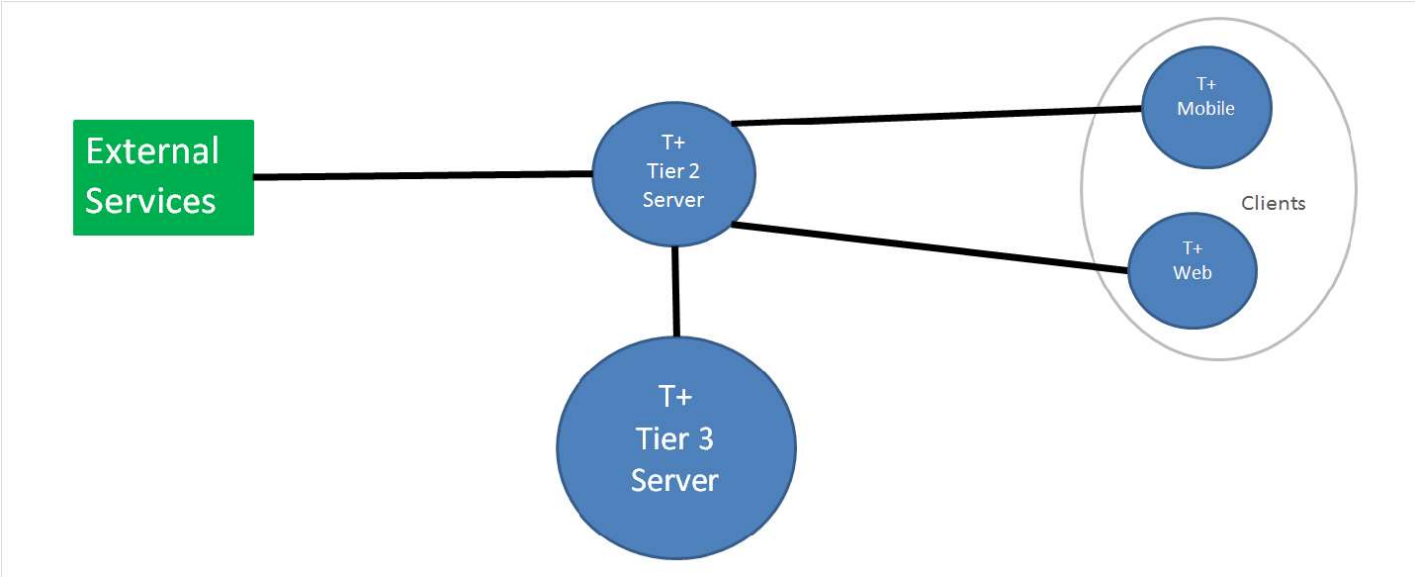
- A **transportation database** component will keep track of the current updated state of all transportation knowledge of the system.
- A **transportation data management** component will provide the same graph abstraction described in the T+ Mobile architecture
- Multiple **API-specific modules** will manage all the needed updates of the data provided by third-party services
- An **ephemeral data component** will manage all temporary data (such as traffic estimates, or sharing vehicle availability)
- A **client feed** component will manage feed request lists by clients to efficiently forward the specific requested ephemeral data updates
- A **network management** component will handle and dispatch all client requests
- An **account data** component will manage all account-related operations and checks by forwarding requests to the Tier 3 centralized server
- An **optimization component** will provide online optimization services in order to enable BMO online calculations without the need to download full maps for occasional use. This module will be a lighter version of that embedded into the T+ Mobile as it will provide solutions on request but then their performance will be kept up to date and supervised by the client itself.

2.A.1 High level components and their interaction

The high level components architecture that make up Travlendar+ system are :

1. T+ Server, Tier 2, (application server, database with geographical data)
2. T+ Server, Tier 3, (database with user data)
3. T+ Clients applications (either mobile or web)

In addition the T+ Server interacts with the external services that provide all the necessary information.



2.A.2 Adopted Technology

Application server (T+ Server, Tier 2 & Tier 3)

The application server on Tier 2 and Tier 3 is going to be written in the **Java** programming language.

Web Server (T+ Server, Tier 2)

The web server on Tier 2 is going to be hosted by **NGINX.**
NGINX is a free, open-source, high-performance HTTP server and reverse proxy, as well as an IMAP/POP3 proxy server.
It is known for its high performance, stability, rich feature set, simple configuration, and low resource consumption.

Web Application back-end (T+ Server, Tier 2)

Since our web backend should be capable of a live feed system toward the browser, neither PHP nor other static programming environments were suitable. For this very reason we decided to work with **Node.js**, a very powerful JavaScript runtime engine.
Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient.
Nowadays Node.js is becoming an industry standard for his capabilities and performances.
It it dynamic in that it can host a powerful WebSocket server and allow a browser that supports HTML5 to open a bidirectional connection for live streams of data.

Web Application front-end (T+ Desktop)

The web application front-end is going to be developed according to the latest available technologies.

HTML5 will structure the web pages, **CSS** will give them a fancy look and **Javascript** is going to bring them to life with dynamic interaction with the Tier 2.

Mobile application (T+ Mobile)

The mobile application is going to be developed in the **Java** programming language (more information are provided in the "Other design decisions" section).

Encryption layer

All communication over the network are going to be encrypted with Secure Socket Layers (**SSL**) to grant a reliable level of security.
Thus socket TCP connections between T+ Mobile and T+ Server will adopt this encryption technique and web pages are going to be served by NGINX using **HTTP Secure (HTTPS),** the SSL implementation for HTTP requests and responses.

Database Manager System (DBMS)

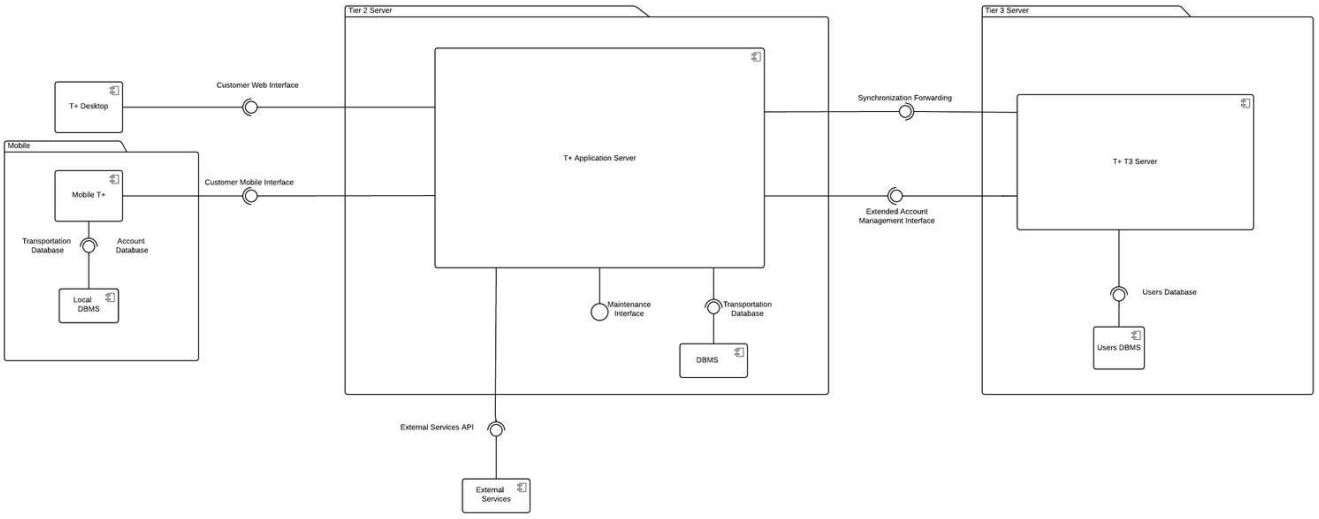
The DBMS for Tier 2 and Tier 3 will be **PostgreSQL.**
PostgreSQL is a powerful, open source object-relational database system. It has been an industry standard for many years and has a strong reputation for reliability, data integrity, and correctness. It runs on all major operating systems, including Linux, macOS and Windows.
It is fully ACID compliant, has full support for foreign keys, joins, views, triggers, and stored procedures (in multiple languages, such as JSON).
Finally it offers native support to many programming languages such as C/C++, Java, Python and lots more.

JDBC Driver

In order to interact with the DBMS within our application we decided to use the native **PostgreSQL JDBC Driver.**
It allows Java programs to connect to a PostgreSQL database using standard and database independent Java code.
Written in Java itself, it communicates with the PostgreSQL native network protocol.

2.B Component view

The following schema represents the architecture of T+ System and the interactions between its components.



T+ Desktop

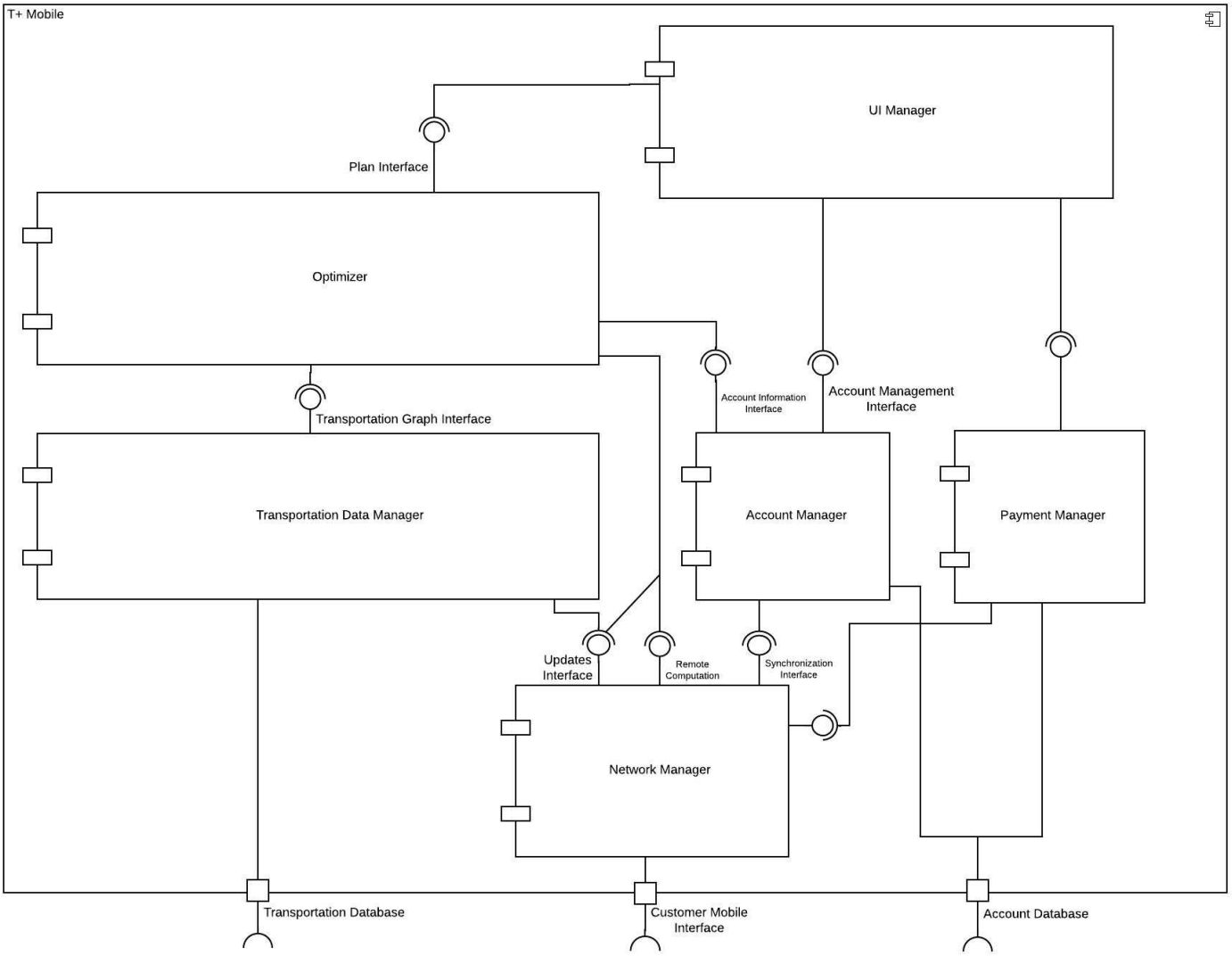
T+ Desktop is meant to provide the user a secondary interface to the T+ system when he prefers to use a desktop application rather than the mobile one to schedule his tasks and edit his preferences. Even though it has less functionalities, it still shares information with the server throughout the network.
Its purpose is to browse and edit tasks, constraints and preferences, so the calendar component is going to be one of its core services, while there will be no need for local transportation data or optimization services.
When the user inputs the data for a new task the BMO option is delayed later when the user will use the mobile application. This is not a limitation because Travlendar+ is designed to help the user during the day giving easy directions that guide to every appointment in time and therefore it is clearly thought to be primarily a mobile application.

T+ Mobile

T+ Mobile is responsible for the communication and interaction between mobile devices and the rest of the system. It uses the custumer mobile interface made available by the T+ Application Server.

This component must be able to adapt to different devices (Android, iOS) using different frameworks and libraries.

The following diagram describes its internal structure and the interaction between its components:



- **Optimizer**
  The optimizer is the core of our T+ system. It computes and updates the BMO for the user taking care of all the data coming from external sources and account preferences and constraints.
- **UI Manager**
  The UI manager sub-component gives the user the possibility to interact with the system in a simple way and allows the user to take advantage of all the features of the application.
- **Transportation Data Manager**
  The transportation data manager is the core back-end of the transportation services. It manages the data for all the transportation means and takes care of the ephimeral data (traffic, meteo, ..).
- **Account Manager**
  The account manager takes care of all user related information such as personal information, preferences, constraints, passes and all his tasks.
- **Payment Manager**
  The payment manager deals with all the purchases made by the user.
- **Network Manager**
  The network manager forwards the T+ mobile client requests to the correct server and it therefore allows all the mobile sub - components to stay up to date.

Client Local DBMS

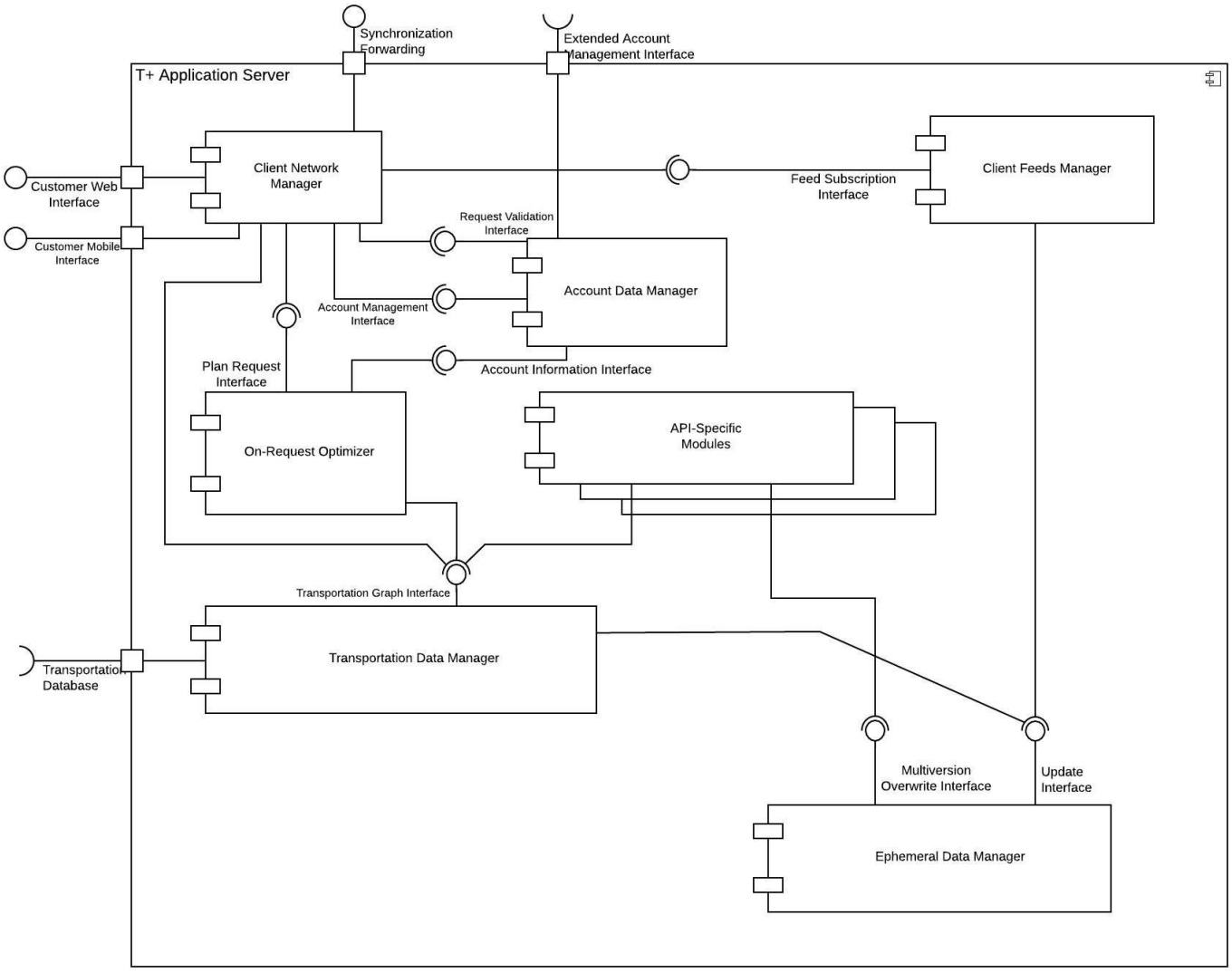
The local DBMS takes care of personal information of the user including credentials, tasks, tickets, constraints and preferences. Since the database should contain very heterogeneous data we opted for PostgreSQL which give the possibility to have either relational tables (account data) or JSON documents(transportation data).

T+ Application Server

The application server should manage all requests from clients and keep the transportation data constantly up-to-date.

It includes dedicated components to manage load-critical tasks such as ephemeral data indexing and specific client update forwarding.

The management of account related tasks is delegated to the tier 3 server component in order to grant security and confidentiality of such data, while transportation data (both static and dynamic) are kept locally to improve access times: it is public data, its protection is not critical.



- **Client Network Manager**
  The client network manager manages client network requests and interactions in general. It also forwards T3 messages to clients.
- **Client Feeds Manager**
  This sub-component takes care of the client feed lists (associations between clients and transportation graph elements that they want to keep up-to-date) and forwards ephemeral data updates only to those clients that subscribed to the subject of that piece of information.
- **Ephimeral Data Manager**

The ephimeral data manager has to handle all the data from API modules about temporarily limited events (such as traffic info, wheather...)

- **API-Specific Modules**
  These modules interact with all the different external APIs in order to gather all the necessary information into our system.
- **On- Request Optimizer**
  This module is a lighter version of the T+ Mobile Optimizer. It can handle BMO computation requests from clients that do not have the transportation data about the requested city, but it cannot take the computed BMO up-to-date.
- **Account Data Manager**
  This component takes care of account-related operations in the Tier2 servers, thus it has to certify user identities and forward and filter requests and change notifications from clients to the Tier3 server.

Transportation DBMS

The transportation DBMS holds the static representation of a number of available cities. In clients it contains all transportation data about all locally downloaded maps, while in application servers it holds all transportation data about the cities under the coverage of that specific server.

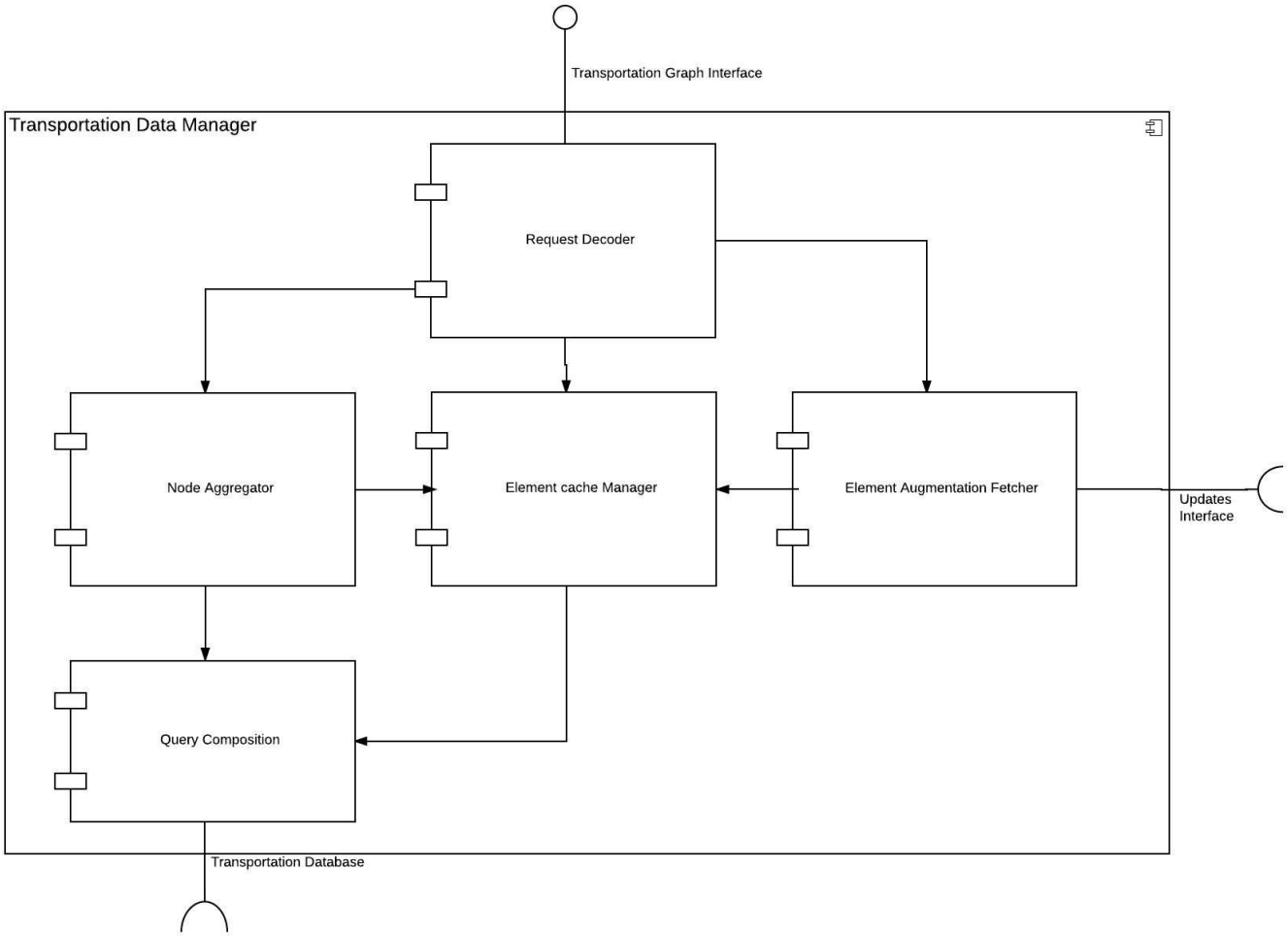
The DBMS must guarantee ACID properties but it must also be able to manage data with more free structures (all the information about transportation means). For this reasons we think that PostgreSQL can be the best choice because it can support a large amount of data types (including JSON).

Transportation Data Manager

Nodes and edges of the transportation graph outside this module will implement their interface by simply delegating every call to the Request Decoder in order to be lightweighted objects even when representing heavy nodes: this mechanism allows the Transportation Data Manager to optimize memory usage even when other components are dealing with a large number of nodes at the same time.

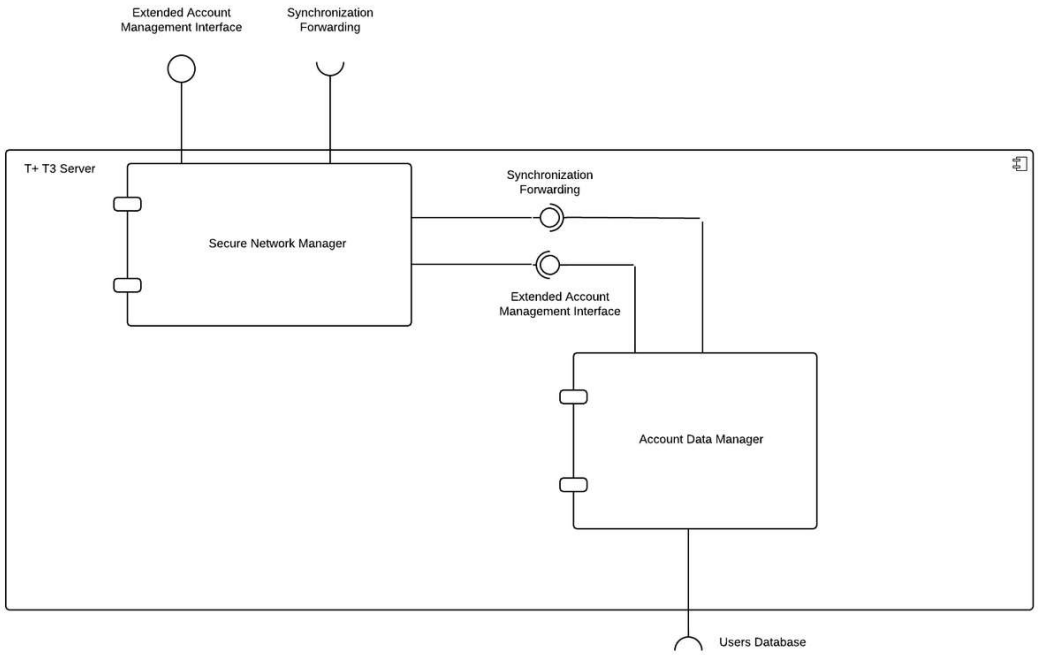
The structure of this component consists of six different modules:

- The **Request Decoder** is the entry point of every call: this will coordinate the activity of all subcomponents in order to fulfill a request
- The **Element cache Manager** is the core of the Transportation Data Manager: it holds all the currently available nodes and edges, indexed by ID, and keeps track of the usage history of the instanced nodes to be able to free memory with the least requested elements when needed. If some element is requested but not available, it will fetch it in the Query Composition module.
  This provides a proxy towards the costly database selection through caching and update through lazy persistence synchronization.
- The **Node Aggregator** module provides the abstraction of geographical coordinate translation into node IDs with different precision settings, enabling the possibility to fetch the nodes into a given coordinate range, aggregate near locations into a single node (essential feature to integrate data from different services into a single coherent model) and create and return new nodes when they are not already present.
- The **Element Augmentation Fetcher** requests and applies ephemeral data to elements in order to dynamically augment the static design of persistent data into almost real-time updated behaviour
- The **Query Composition** module standardizes the queries that are made to the Transportation Database in order to enforce that they respect specific patterns to ensure optimization.



T+ Tier3 Server

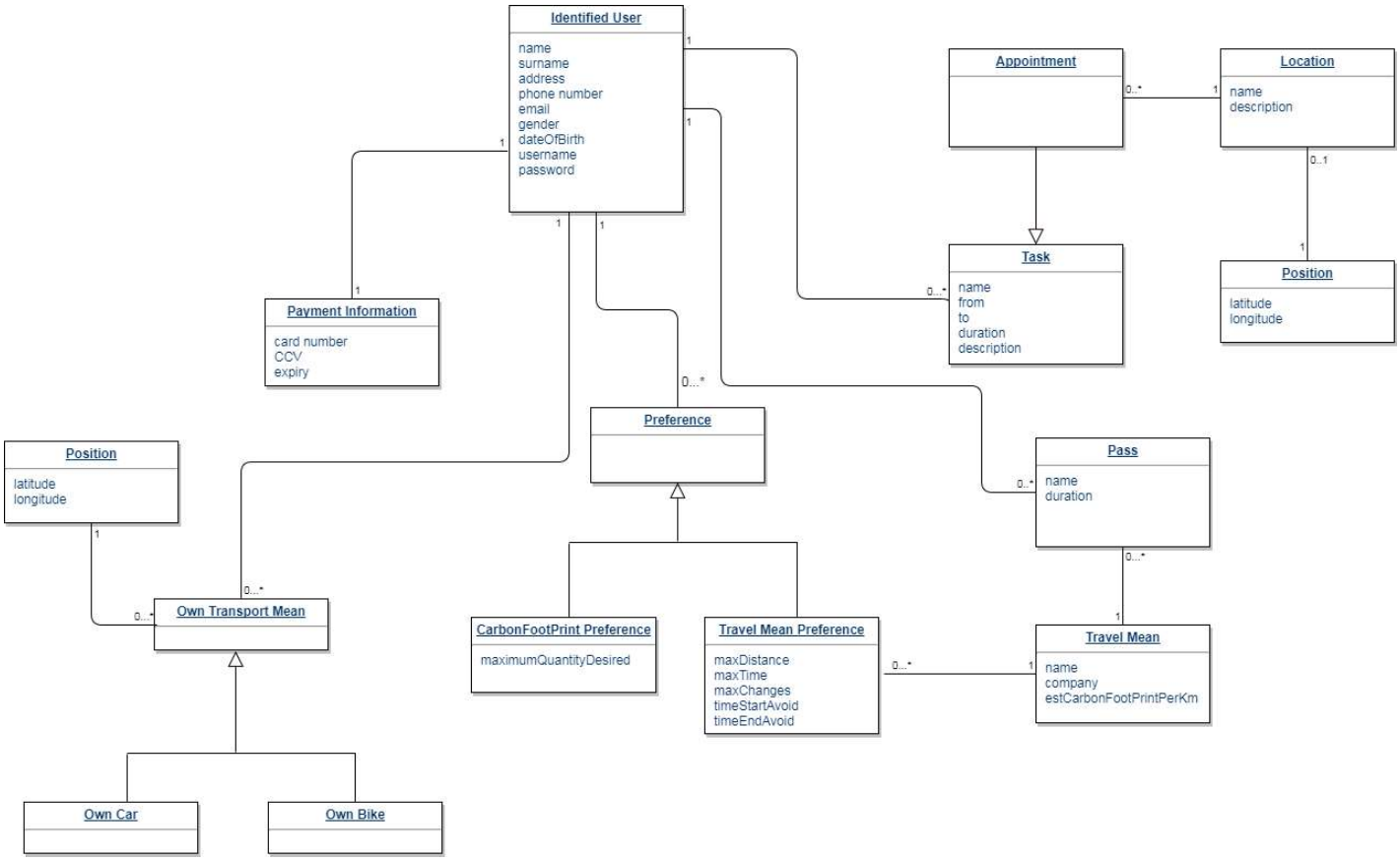
The Tier3 Server is designed to provide centralization and protection to user related data.
The network layer interacts only with the application servers in order to protect data from external attacks and forwards filtered requests to a simple logic layer that implements automatic account synchronization between devices and select/update queries



- **Secure Network Manager**
  The secure network manager is the most important sub-component of the tier 3 server because shares data of the users with either the mobile\web client or the application server in a safe way.
- **Account Data Manager**
  Interact with the database in order to access and store all the data.

Database Structure

A possible structure for the **user information database** is the following:

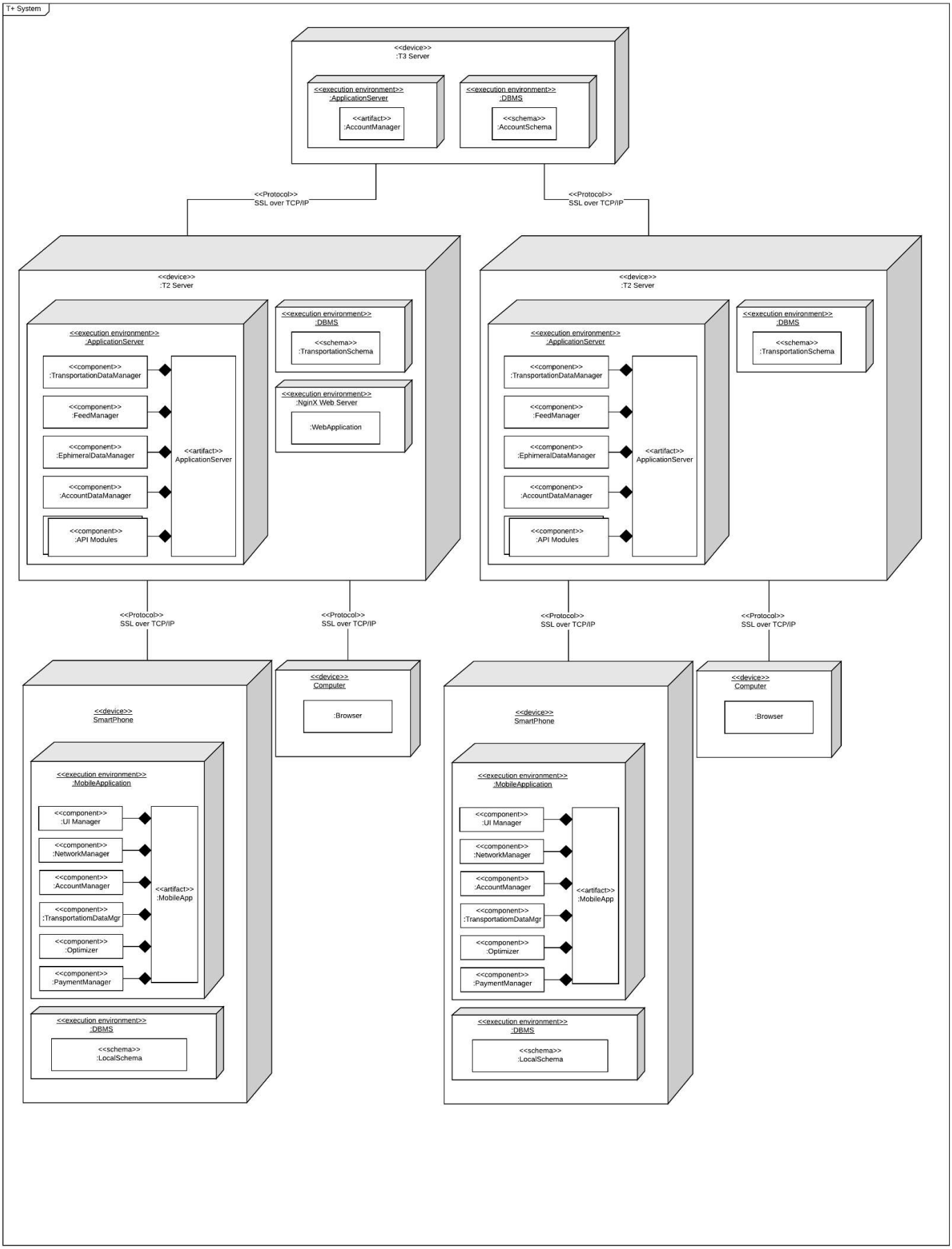


**position is showed two times only for graphical issues.

The Tier 3 database has a similar structure with the difference that the payment information are not present while it has all users informations unlike local database.

Geographic data management is definitely one of the most important challenges in our system. The interaction with multiple APIs is essential. All the informations stored have to have a standard structure. We thought of two main tables: nodes table and arcs table. The nodes table has two fields (ID, JsonNodeField) while the arcs table has 4 fields(IDstart, IDend, IDmean, JsonArcsField). In the Json fields we can store all the possible heterogenuous informations that an interaction with multiple API can produce. The ID of a node depends exclusively on its geographic coordinates(latitude,longitude) with a granularity of 5-10 meters.

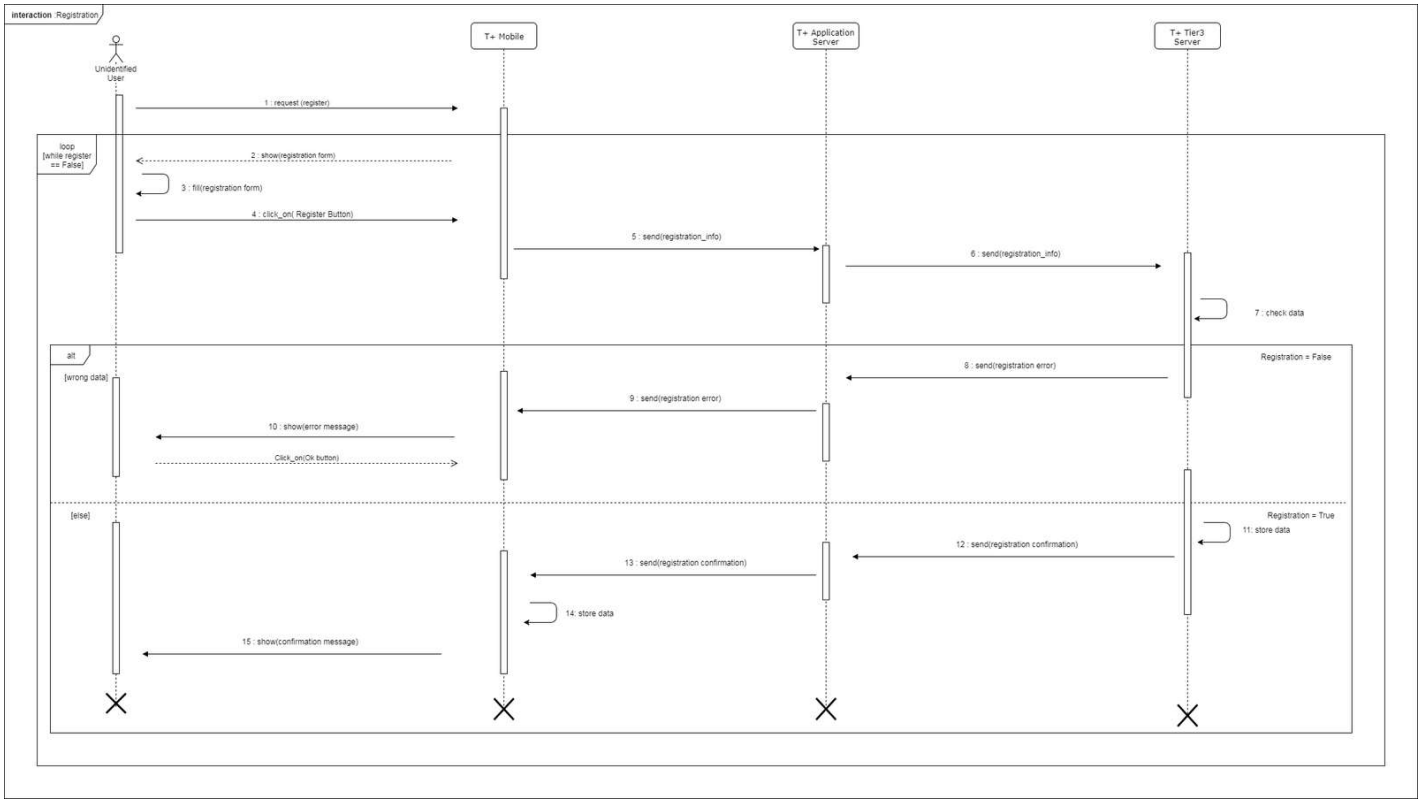
2.C Deployment view



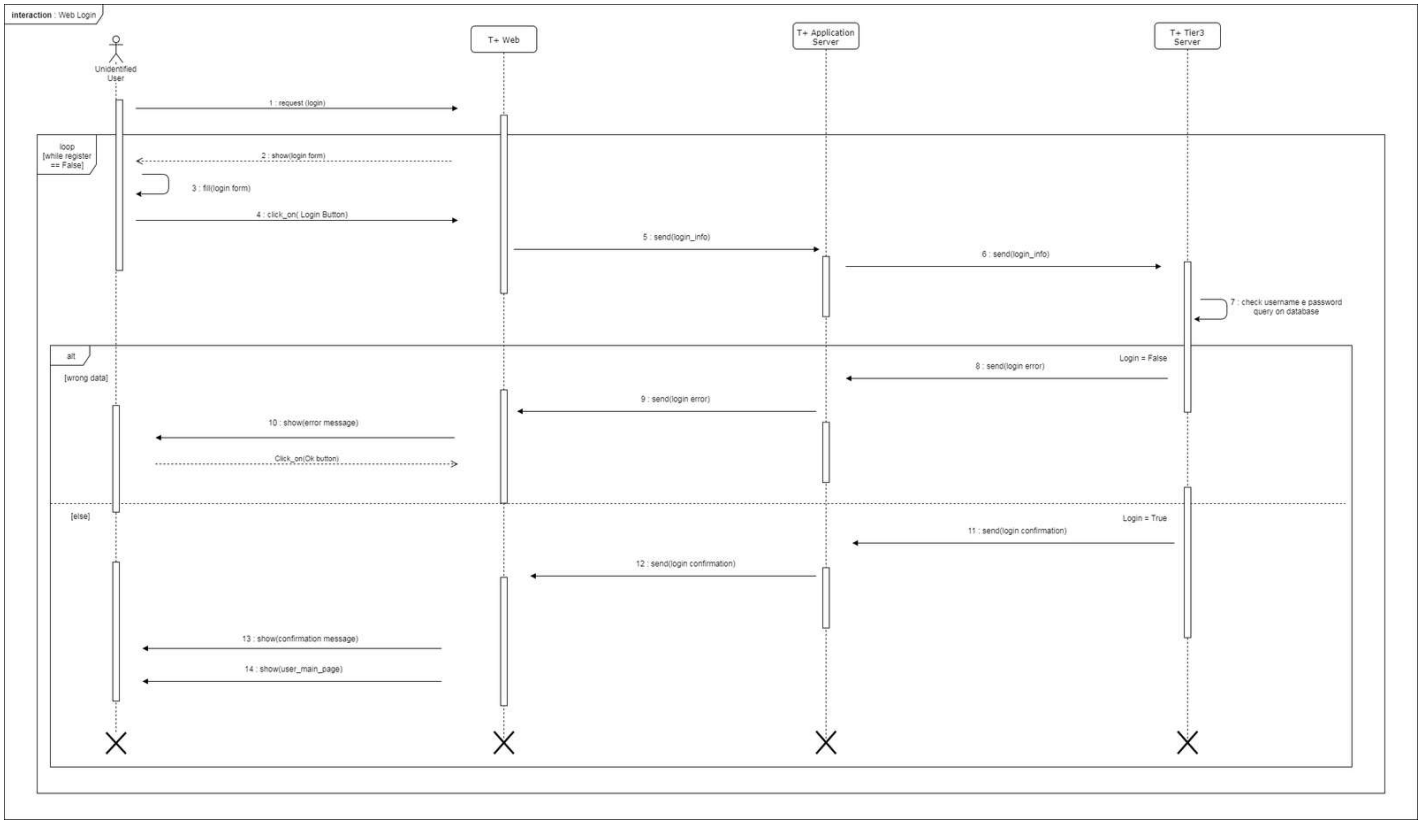
2.D Runtime view

The main purpose of this section is to show the relationship and interactions between the components of the system during some real scenarios.

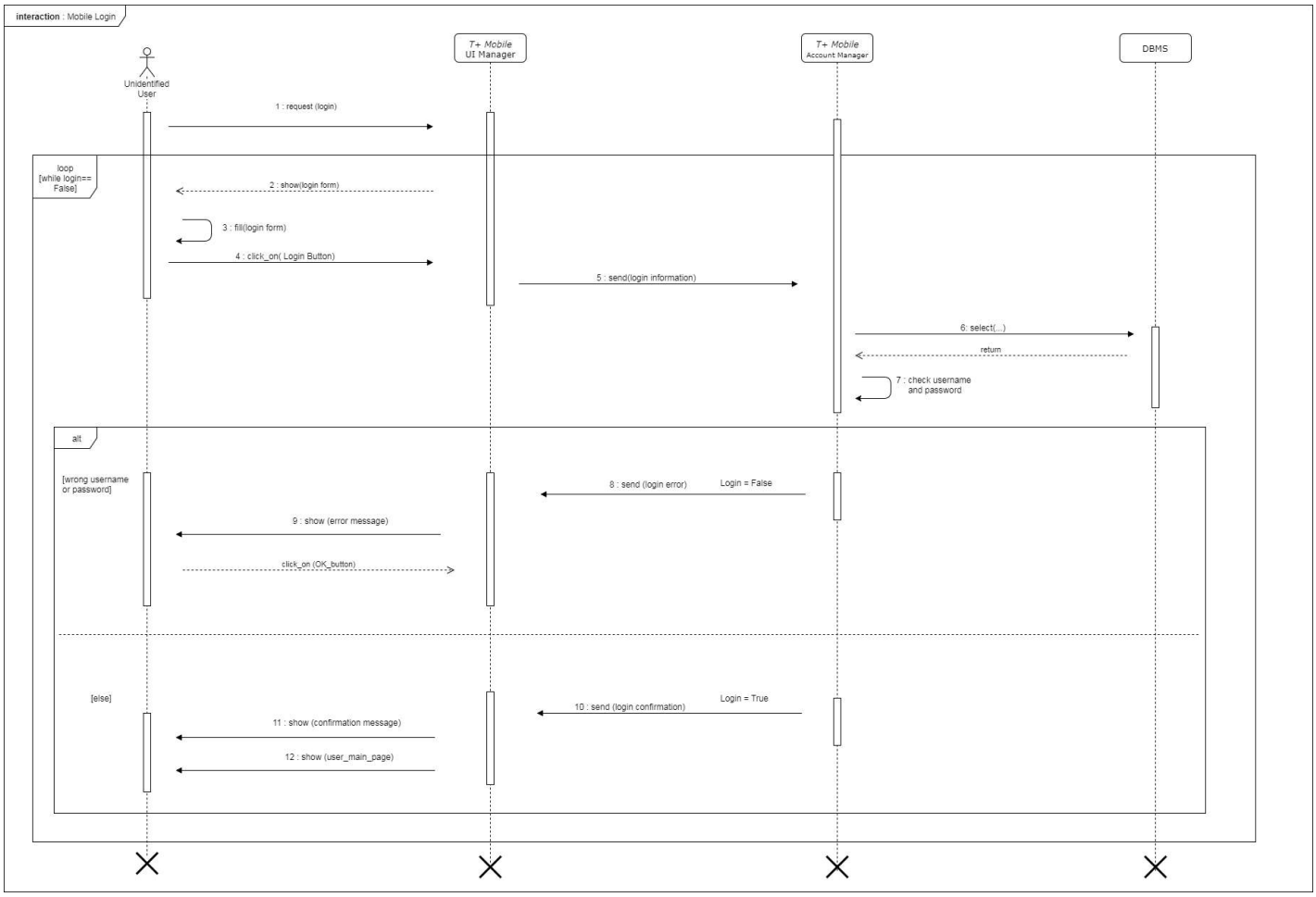
2.D.1 Registration



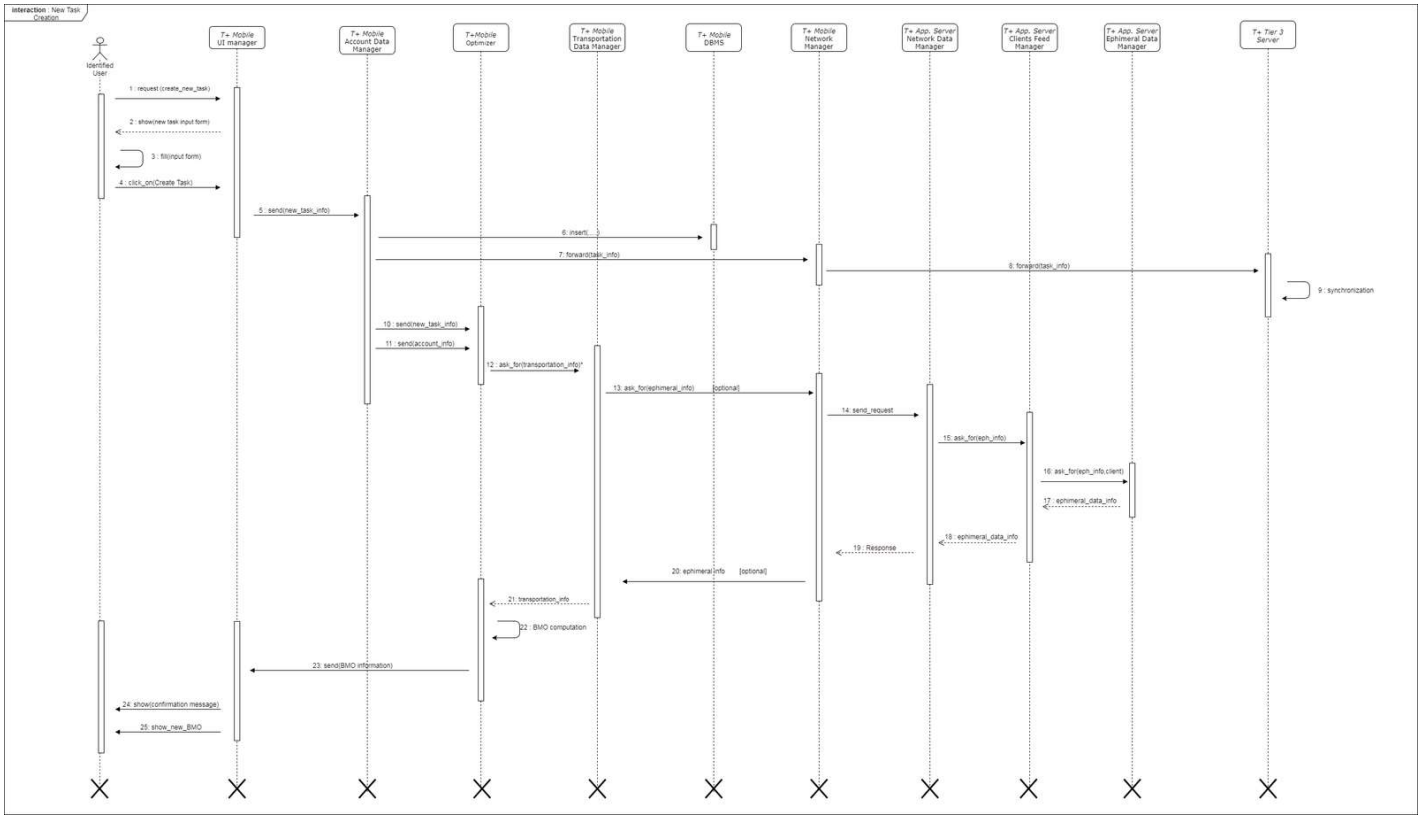
2.D.2 Web Login



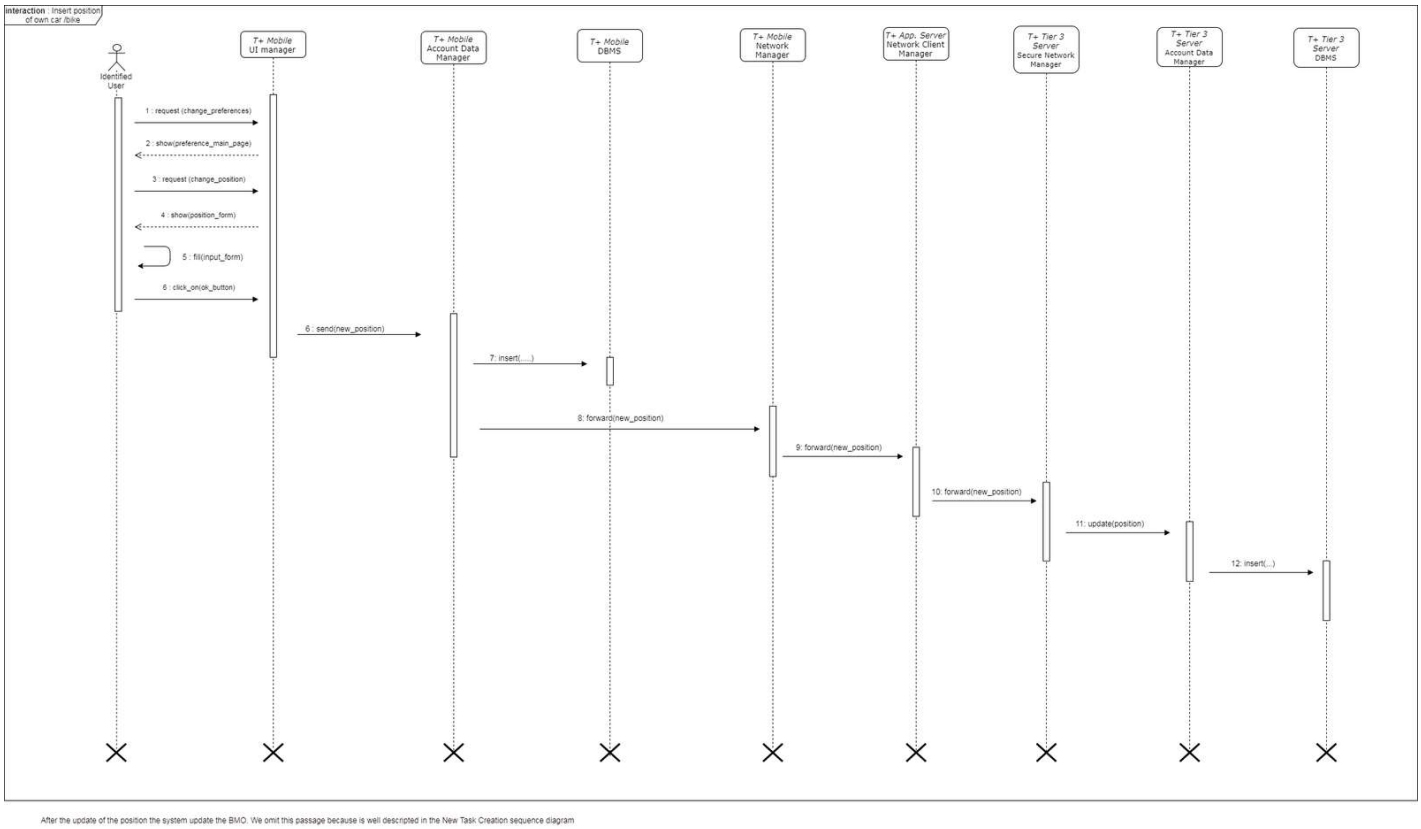
2.D.3 Mobile Login



2.D.4 New task creation



2.D.5 Update position of own car/bike



2.E Component interfaces

This section gives a brief description of all the interfaces that Travlendar+ communicates with.

2.E.1 Customer Mobile \ Web Interface

Method	Parameters	Functionalities	Responsible sub-component
register()	name : String surname: String address: String phone number: String email: String gender: String date_of_birth: String username: String password: String	Register a new user and create a new account	Account Data Manager
login()	username: String password : String	Check username e password of the user	Account Data Manager
logout()	username : String	End the user session	Account Data Manager
updateAccount()	username: String new address: String new phone number: String new email: String new password: String	Update some of the user's information	Account Data Manager

deleteAccount()	username : String	Delete the account	Account Data Manager
getSelectedTransportationInfo( )	username : String	Get updated information about transportation means, traffic, weather. It gives only the 'news' which are interesting for the users's tasks.	Transportation Data Manager, Ephimeral Data Manager
addPass()	username: String pass : Pass	Register a new pass for the user (with all the informations about transport mean, duration, price, ecc)	Account Data Manager
addPreference()	username: String preference: Preference	Add a new preference for that user	Account Data Manager
changePreference()	username: String old Preference: Preference new preference: Preference	Change an user preference	Account Data Manager
deletePreference()	username: String preference: Preference	Delete an user preference	Account Data Manager
deletePass()	username: String pass: Pass	Delete a pass previusly registrated	Account Data Manager
addTask()	username: String task: Task	Add a new task	Account Data Manager
changeTask()	username: String old Task: Task new Task: Task	Change a task	Account Data Manager
deleteTask()	username: String task: Task	Delete a task previusly added	Account Data Manager
pastTasks()	username: String	Old tasks information	Account Data Manager
synchronize()	username: String	Synchronize the local database with all the updated information (either from the application server or the tier 3 server)	Account Data Manager + Transportation Data Manager

2.E.2 Extended Account Management Interface

Method	Parameters	Functionalities	Responsible sub-component
getUserInformation()	username : String	Get user account information	Account Data Manager
getUserToDoTask()	username: String	Get user incoming tasks	Account Data Manager
getUserPreferences()	username: String	Get user preferences	Account Data Manager
getUserPasses()	username: String	Get user passes	Account Data Manager

2.E.3 Synchronization Forwarding

Method	Parameters	Functionalities	Responsible sub-component

synchronizeUserInformation()	username : String	Get the user account information coming from the mobile application	Client Network Manager
synchronizeUserToDoTask()	username: String	Get user incoming tasks	Client Network Manager
synchronizeUserPreferences()	username: String	Get user preferences	Client Network Manager
synchronizeUserPasses()	username: String	Get user passes	Client Network Manager
synchronizeAll	username: String	Get all the user data	Client Network Manager

2.E.4 Mobile T+ Internal Interfaces*

Method	Parameters	Functionalities	Interface	Sub-component
getBMO()	#######	Get user BMO	Plan Interface	Optimizer
getUserInformation()	#######	Get user information	Account Management Interface	Account Manager
login()	username: String password: String	Check username e password of the user	Account Management Interface	Account Manager
getTasks()	#######	Get user tasks	Account Management Interface	Account Manager
addPreference	#######	Add a new preference for that user	Account Management Interface	Account Manager
getPreferences()	#######	Get user preferences	Account Information Interface	Account Manager
updateTransportationInfo()	#######	Update transportation info of the local database	Updates Interface	Network Manager
synchronizeAccount()	#######	Synchronize account data	Synchronization Interface	Network Manager
pay()	#######	Pay all hanging bills	Payment Interface	Payment Manager

*only the most relevant functionalities

Transportation Graph Interface
The Transportation Graph Interface is a particular interface that need special attention for its complexity and importance. This interface provides to the upper modules a common and powerful abstraction to interact with the transportation data stored in T+ system.
The interface is intended to represent the abstraction of a transportatio graph, where nodes correspond to geographical locations and edges correspond to possible movements with different transportation means (two nodes can be connected by more than one edge).
Each graph is associated to a covered city and its peripheral areas, and one more graph is dedicated to international transportation where each node is linked to some city node.

The main interface should provide the following methods:

- getCityGraph(): given the name of a valid covered city, it returns the graph corresponding to that city. A reserved name is used for the international transportation graph

The graph interface should provide the following methods:

- getNode(): given geographical coordinates, it returns a node corresponding to them
- getExistingNodesInRange(): given a geographical coordinate range, it returns all known nodes inside them

The node interface should provide methods such as:

- getForwardStar(): returns all outcoming edges of this node
- getBackwardStar(): returns all incoming edges of this node
- addEdge(): inserts a new transportation edge into the graph (the ending node may be specified either explicitly or by geographical coordinates)
- getConnectionNode(): in the international graph this provides the corresponding city and the corresponding gate node in that city; in a city graph it returns if the corresponding international graph node linked to this node, if any

The edge interface should provide methods such as:

- getStartingNode(): returns the starting node of this edge
- getEndingNode(): returns the ending node of this edge
- computeForwardState(): for BMO forward computation, given an initial path state it computes the next path state given that its transportation mean is used (see section 3 about Algorithm Design)
- computeBackwardState(): for BMO backward computation, given a final path state it computes the starting path state given that its transportation mean is used (see section 3 about Algorithm Design)

Note that transportation information about nearby places may come from different sources not knowing each other and may be submitted in our system independently, for this reason it is not possible to explicitly instanciate a new graph node: node addition is only possible through the methods getNode() and addEdge() when specifying the ending node by its coordinates. The Node Aggregator component in the Transportation Data Manager should evaluate whether a new node should be created into the database or a nearby node should be picked instead.

It is foundamental that nodes and edges obtained through this interface are no more than stubs: they must be lightweighted objects that forward method calls to the Transportation Data Manager in order to have full control over memory consumption and database synchronization even when the upper layers need to manipulate many nodes at the same time.

2.E.5 Application Server Internal Interfaces*

Method	Parameters	Functionalities	Interface	Sub-component
getBMO()	username: String	Get the user BMO	Plan-Request Interface	On-Request Optimizer
getUserPreferences()	username: String	Get the user preferences	Account Information Interface	Account Data Manager
login()	username : String password: String	Performs the login	Request Validation Interface	Account Data Manager
getUpdatedTransportationInfo()	username: String	Get the updated transportation information for that specific client	Feed Subscription Interface	Client Feeds Manager
getEphimCityTransportationInfo()	city: String	Get the updated information for that specific city	Update Interface	Ephimeral Data Manager

*only the most relevant functionalities

2.E.6 Tier 3 Server Internal Interfaces*

Method	Parameters	Functionalities	Interface	Sub-component
getUserInformation()	username: String	Get user informations	Extended Account Management Interface	Account Data Manager
getUserPreferences()	username: String	Get user preferences	Extended Account Management Interface	Account Data Manager
getUsertTasks()	username : String	Get user tasks	Extended Account Management Interface	Account Data Manager
addTask()	username: String task: Task	Add a task to that user	Extended Account Management Interface	Account Data Manager
addPreference()	username: String preference: Preference	Add a preference to that user	Extended Account Management Interface	Account Data Manager
synchronize()	username: String	Synchronize the information for that user with the mobile ones	Synchronization Forwarding	Secure Network Manager

*only the most relevant functionalities

2.F Selected architectural style and patterns

Model View Controller (MVC)

The overall architecture of both T+ Mobile and Desktop lends itself well with the MVC (Model-View-Controller) design pattern. This pattern exhibits extreme flexibility because the view doesn't care about the underlying model and allows for future implementation and enhancements.
Being able to tease these components apart makes the code easily reusable and independently testable. Furthermore MVC grants high cohesion: it enables logical grouping of related actions on a controller together. The views for a specific model are also grouped together.
Moreover, the MVC separation allows different services to be run and managed more easily on distributed architectures, allowing for better scalability of the server-side system.
For this reason the architectural components themselves are going to be subdivided in one of the three levels as follows:

- **Model**
  - Client local database component.
  - Server and client transportation database component
  - Server ephemeral data component
  - Server account data component
- **View**
  - Client UI Management component.
- **Controller**
  - Client and server transportation data management component.
  - Client and server optimization component.
  - Client calendar component.
  - Client network interface component.
  - Server network management component.

- Client preference management component.
- Server API-specific components
- Server client feeds component

Observer

The T+ System will rely highly on an event driven architecture.
For this very reason the Observer pattern is going to take care of the dispatch of messages and data between Server, Mobile and Desktop applications.

Specific observer interfaces are going to be designed and implemented by T+ Mobile and Desktop to allow event subscription and un-subscription whereas T+ Server is going to fire these events when required, such as to feed the clients with new data.

On the server side, this service will be managed by the client feeds component that will group all client subscriptions to events related to specific arcs of the graph, indexed by arc-id. Every time new data is available about a specific arc, this will be forwarded as an event to all clients related to that arc, if any.

On the client side, the optimization component should keep the current BMO time estimation up-to-date, thus it will request a feed subscription only to the related graph arcs in order to provide a lightweighted real time estimate of the arrival time.

We wanted to abstract as much as possible from the lower levels of communication and provide a highly scalable and versatile interface to dispatch events, no matter the underlying network architecture (Bare TCP/IP sockets, Datagrams or RMI).

Adapter

Due to the variety of API providers that T+ Server is going to interact with we decided to adopt an adapter design pattern. Since many API are HTTP-Request based whereas other expose different approach methodologies it is mandatory to create a unified interface to interact with them. Thus the modules that are going to interact with the different APIs will interact with semi autonomous components in a simple manner. Furthermore the adoption of this adapter design will grant high scalability in terms of adding and removing API providers.

Every API adapter should interact with the data provider and update the static graph or the ephemeral data through the transportation data management component and the ephemeral data component. Other components of the system will never have to interact directly with them, as all provided data will be embedded into the transportation graph.

Decorator

Real world transportation means can change the service they provide during time due to several possible contingent factors (such as traffic may make a road slower, or strike may make some lines temporarily unavailable). As T+ system is supposed to virtually support all possible transportation means and their temporary variations, we need a way to apply any kind of effect on any kind of transportation edge. The decorator pattern applied on transportation edges and ephemeral data allows us to dinamically change the external behaviour of all transportation arcs based on last-minute data.

Ephemeral data will take the shape of decorators for a specific edge of the transportation graph, allowing the transportation data manager component to query for related decorations before returning a specific edge. This mechanism will be completely transparent to upper layers interfacing with the transportation data manager.

2.G Other design decisions

Programming environment

Our software is going to be written in the Java programming language for a few reasons:

- It is one of the industry standards.
- The language has great framework support.
- The Android SDK is written in Java and therefore we are going to have cutting-edge performance on that platform.
- With tools such as Codename One cross platform compilation for iOS is going to be accomplished with the same source code written in Java.

Project manager

To embrace industry needs and standards we decided to use Maven as project management tool. This way all libraries and dependencies will be managed by a single component granting smooth building, testing and portability across different IDEs and programming environments.

3. ALGORITHM DESIGN

BMO computation

In order to grant directions to our users, every client must be able to compute BMOs through their optimization component.
The BMO must be calculated differently depending on the situation: if it is an anticipation of the plan of the day the arrival time is known, and the BMO must be computed backward to set a starting time, but if it is a last-minute computation then the starting time is known only and a forward computation must be performed.

Forward BMO computation
An initial path-state is set to the current position of the user, corresponding to the transportation graph node nearest to the given coordinates. The path-state will include the current time and position, the currently used personal or shared mean (car, bike or none), the valid tickets and passes currently available to the user, the path cost corresponding to the specified user preference trade-off (initialized to zero) and the current target list (initialized to the first appointment only)
From each node we can get the forward star of available actions (each one corresponding to a travel mean) and to each action we can ask what would be the final path-state, given the current path-state.
With these conditions, an instance of the A* algorithm can lead to a specified goal, provided that there is some heuristic. If there are no precomputed heuristics available, the default one is used.
The goal of the A* will be set as the first appointment to be reached, and as soon as the first appointment is reached, the forward computation ends. If there are more appointments, then a backward computation is run setting the estimated starting position to the arrival position and the minimum starting time to the arrival time plus the current appointment duration.

Backward BMO computation
An initial path-state is set to the position of the last covered appointment, corresponding to the transportation graph node nearest to the given coordinates. The path-state will include the final position and fixed arrival time, all personal mean types possibly in use to arrive to that point (combinations of car, bike or none, initialized to {car, bike, none}), the valid tickets and passes available to the user or that should be available to the user to arrive at that point, the path cost corresponding to the specified user preference trade-off (initialized to zero) and the current target list (initialized to the appointment list in decreasing order of ending time, without the last ending one and with the estimated starting position appended)

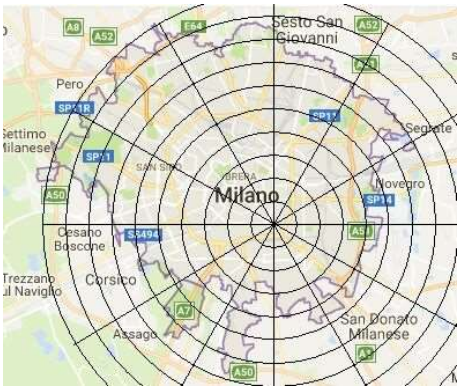
From each node we can get the backward star of actions ending there (each one corresponding to a travel mean) and to each action we can ask what would be the starting path-state, given the final path-state.
With these conditions, an instance of the A* algorithm can lead to a specified goal, provided that there is some heuristic. If there are no precomputed heuristics available, the default one is used.
The goal of the A* will be set as the first goal in the target list, and as soon as one goal is reached, it is popped from the list and the next goal is considered until the list is empty.

Default heuristic

The default heuristic is a very imprecise but consistent estimation of the preference relative cost to go from one point to another. From the set of all transportation means possible in the considered city, the one with the best preference-relative cost per unit of distance covered is chosen and the estimated preference-relative cost is the cost rate of that mean multiplied by the euclidean distance between the two points.

Precomputed heuristic

Performance is a key point in the computation of the BMO. For this reason, a precomputed heuristic on the graph should be implemented to allow faster convergence of the A* algorithm, eventually accepting a pseudo-optimal solution.
The heuristic should be based on precomputed costs based on the constraints and on the preferences of the user itself, for this reason it will be prepared and updated every time the user changes constraints and preferences with low priority over other tasks. This process should also be separable in consistent independent steps in order to be stopped at any moment for higher priority tasks.
The data this heuristic computes are relative to a small fixed subset of the nodes of a city, from now on we will call them heuristic nodes. Heuristic data about all of them can be computed independently, thus the process of heuristic updating can be interrupted at several middle points.
For each heuristic node we want to have a function assigning an estimated cost (relative to the optimization preferences of the user) to reach a given point in space starting from that node.



Clustering of costs based on distance and angle. Elements of the cost matrix correspond to intersections.

In order to build such function we need to compute a matrix of relative point cost estimations: every element E[i][j] in this matrix contains the estimated cost to go from the heuristic node to a point with a fixed distance (given by index i) and in a given geographical angle (given by index j). In this way we are clustering locations around the heuristic node based on distance and angle as in the image on the left.
In order to populate the cost matrix we need to compute the costs of the optimal paths from the heuristic node to each node in the graph (we can use dijkstra's algorithm for this task). Every cost computed will be added in the four nearest cluster points with the following procedure.

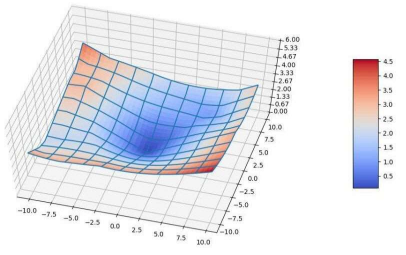
Consider we have calculated the cost associated to point P with respect to heuristic node H as c(P), then let:

- r1 be the matrix index for R1: the maximum R such that distance(P, H)>=R
- r2 be the matrix index for R2: the minimum R such that distance(P, H)<R
- a1 be the matrix index for A1: the maximum A such that angle(P, H)>=A
  a2 be the matrix index for A2: the minimum A such that angle(P, H)<A
-
- p=(distance(P, H)-R1)/(R2-R1)
- q=(angle(P, H)-A1)/(A2-A1)

then we will assign:

- weight p*q to cost c(P) into E[r2][a2]
- weight (1-p)*q to cost c(P) into E[r1][a2]
- weight p*(1-q) to cost c(P) into E[r2][a1]
- weight (1-p)*(1-q) to cost c(P) into E[r1][a1]

clearly E[i][j] will result in a weighted sum of all the reaching costs of nearby nodes.
Now that the cost matrix E is computed, the heuristic cost to go from H to P h(H, P) can be estimated using the same procedure described now, by weighting the estimations of the four point costs around.



A qualitative visualization of the shape of the heuristic h(H, P).

The resulting heuristic function is a non-negative, continuous, locally quadratic function estimating the cost to go from its origin H to a given point P in space, given its coordinates. Once the matrix is given, it can be computed very fast. A qualitative graph of the function in the (x, y) coordinates of point P is given on the left; it has been obtained by a 6x6 cost matrix with two cheap areas around. Such scenario can be found, for example, with time-saving preferences by car near faster roads, or with cost-saving preferences along the routes of cheaper means of transportation.

In order to use this heuristic, we shall now define h(P1, P2) where P1 and P2 are two non-heuristic nodes with given geographical coordinates. Let H be the heuristic node nearest to P1, then we can define: h(P1, P2) = (h(H, P1)^2 + h(H, P2)^2 -2*h(H, P1)*h(H, P2)*cos(angle(P1-H-P2)))^(1/2) which is the cosine theorem applied to distances radially transformed according to the user's optimization preference.
A geometrical interpretation of this definition of heuristic is the following: consider P1, P2 and H in their geographical positions, and then apply a radial transformation centered in H that transforms every point (x, y) by a radial dilatation by its heuristic cost value h(H, (x, y)), including P1 and P2. After this transformation, the defined heuristic will pull the A* algorithm to search first in the straight line on the resulting space. This means that A* will first explore paths that satisfy a trade-off between making h(H, P1) and h(H, P2) nearer in expected cost (and thus reasonably nearer to each other in terms of path cost) and nearer in angle.

It can be easily seen that this heuristic can overestimate path costs, and for this reason it does not guarantee optimal solutions with A*. A convex combination between this heuristic and the default one can be a parameterized solution to choose between speed and accuracy of the proposed BMO, making the behaviour of the algorithm as more explorative as the contribution of the admissible heuristic is relevant.
In order to further optimize the underestimating default heuristic, we can take the maximum between several different admissible heuristics, obtaining for sure another admissible heuristic that takes the best from every component.
An example of heuristic to combine with the default one is the following: h(P1, P2)=|h(H, P2)-h(H, P1)|. Assuming that discretization error is neglibile, we can consider that h(H, P) for every point P is equal to the real optimal cost k(H, P) by construction. Being k an optimal cost, we have k(H, P2)<=k(H, P1)+k(P1, P2), which is: the optimal path between two points is shorter than or equal to the optimal path between the two points with some other intermediate step to touch. This means that k(P1, P2)>=k(H, P2)-k(H, P1) = h(H, P2)-h(H, P1) and the same holds inverting P1 and P2 (and assuming that k(P1, P2)=k(P2, P1)), so h(P1, P2)=|h(H, P2)-h(H, P1)| is an admissible heuristic.
This last heuristic cannot be used alone as it can output the value of zero even between points distant in space, bot in some cases it gives a higher and better estimation than the default one, thus increasing performance without losing in precision.

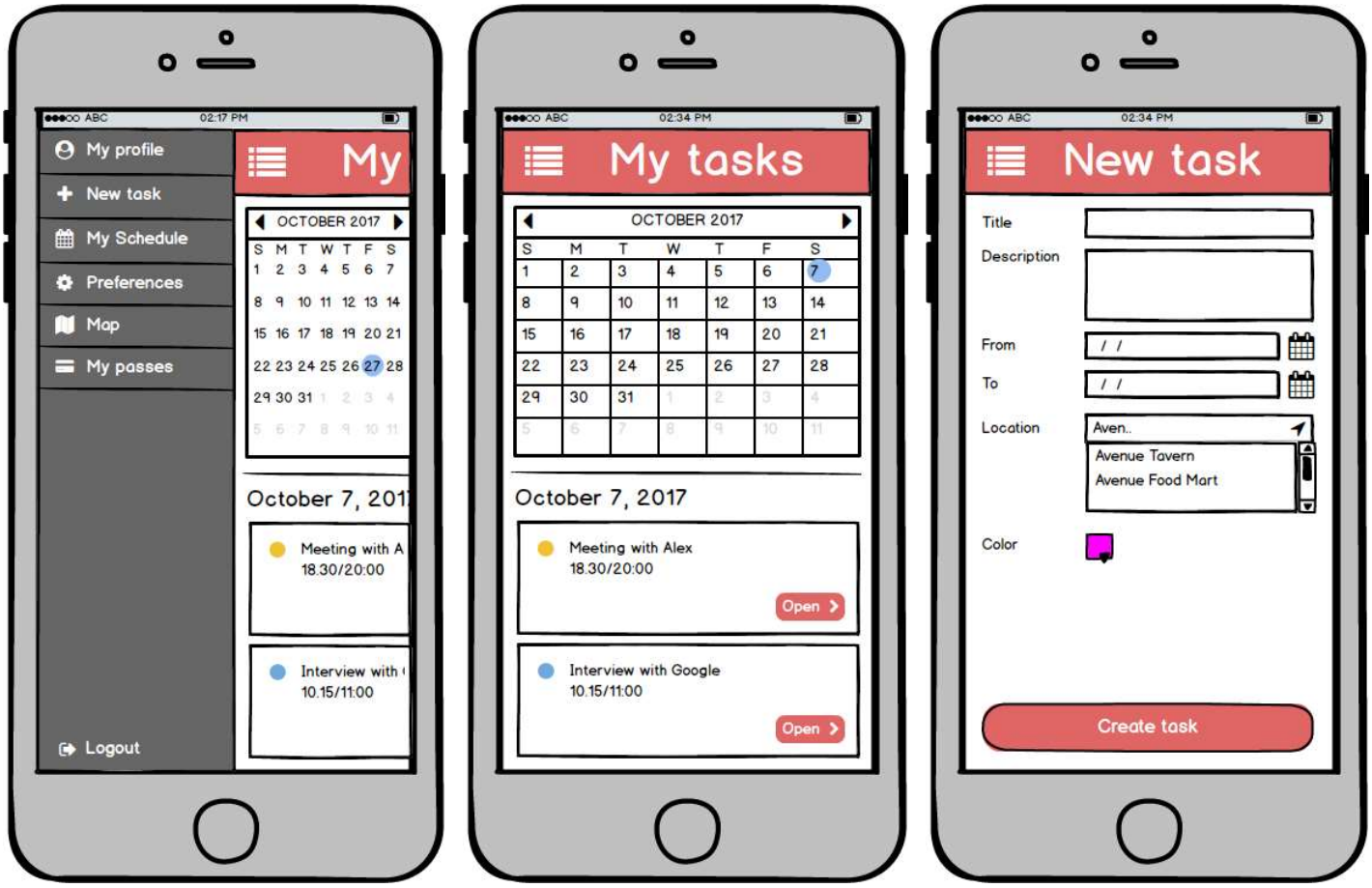
4. USER INTERFACE DESIGN

4.1 Mockups

4.1.A T+ Mobile

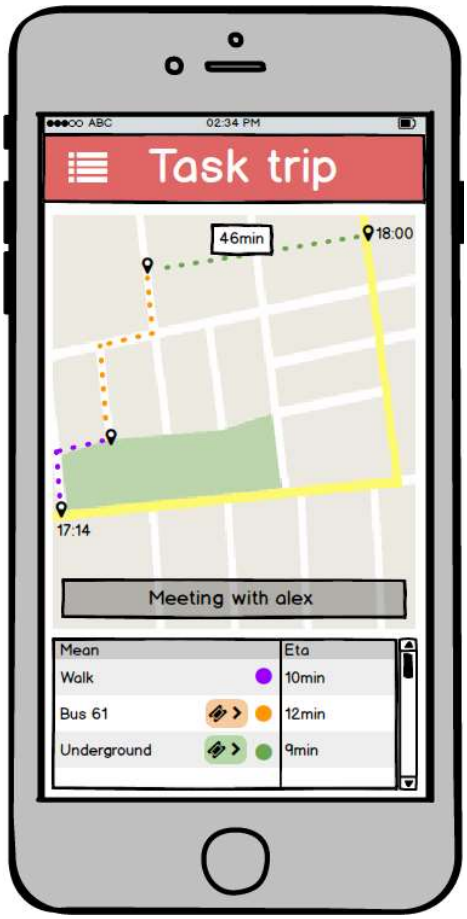
Even though all the mockups could be found in the RASD, some of them have been inserted even here.

Initial view *Registration process*

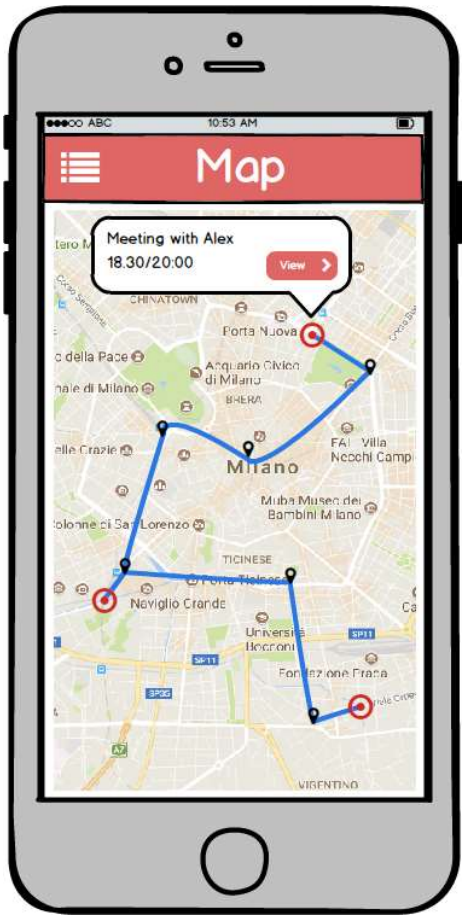


Navigation menu *Calendar view* *New task view*

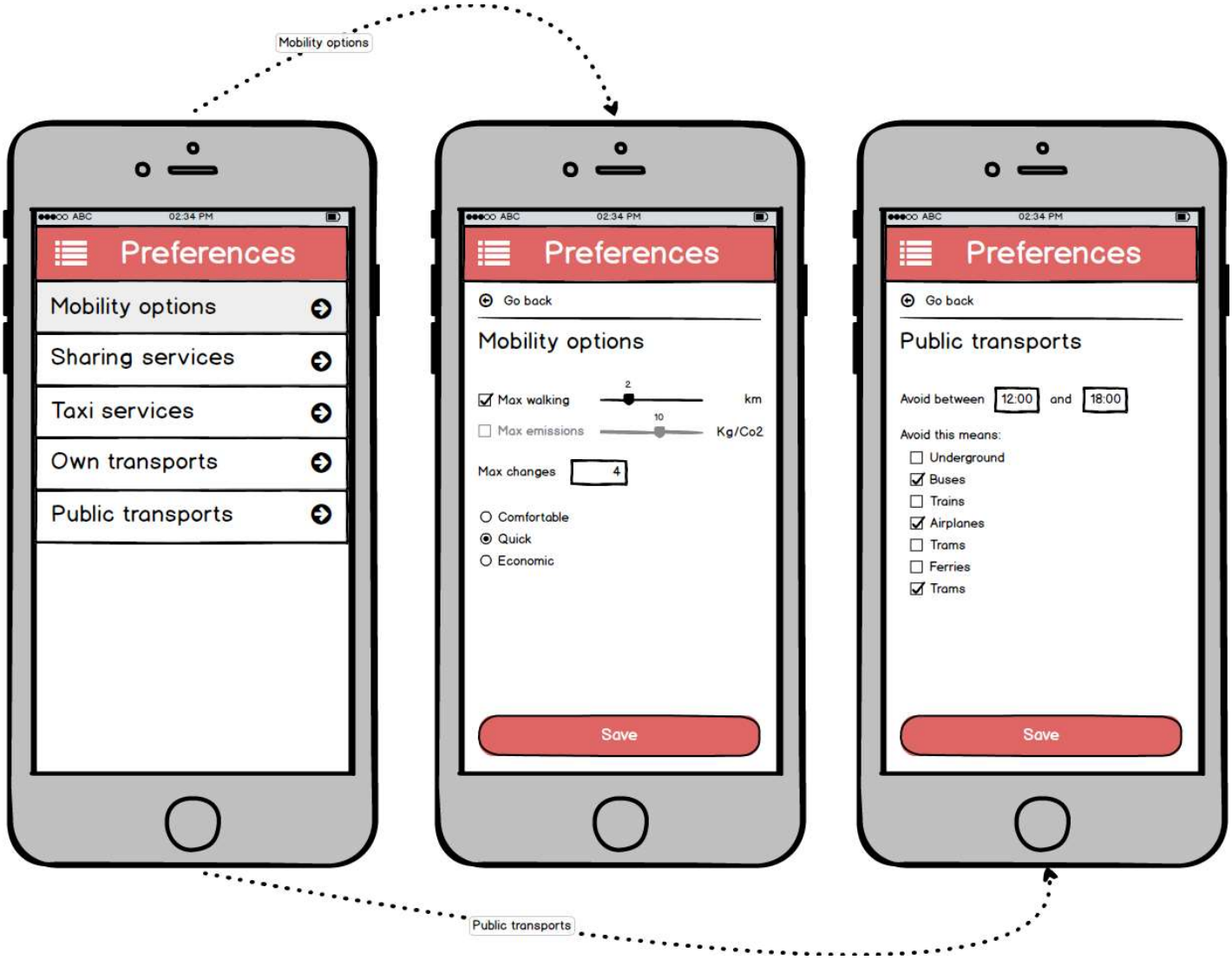
Edit task view



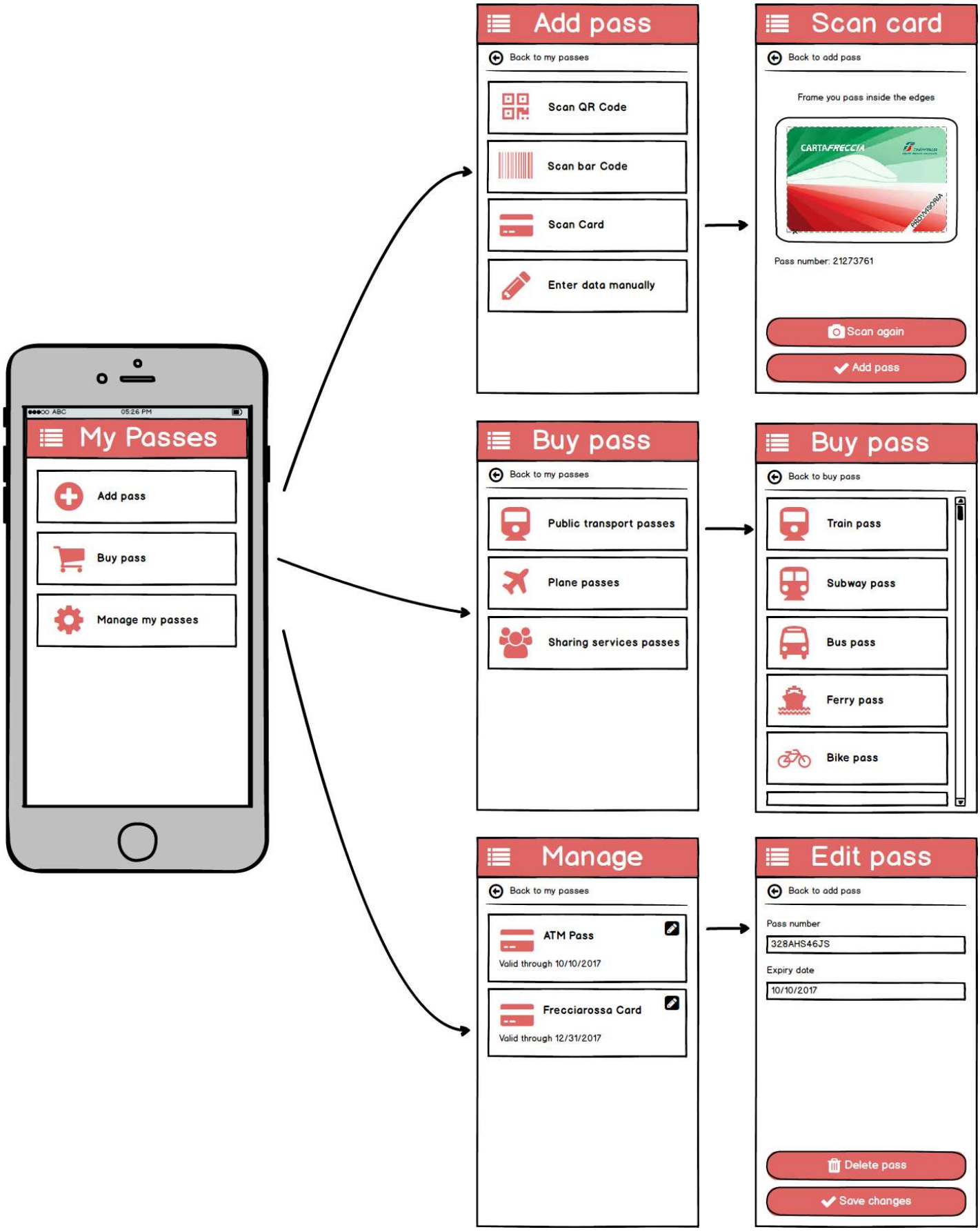
Single task trip view



BMO for task in the time coverage

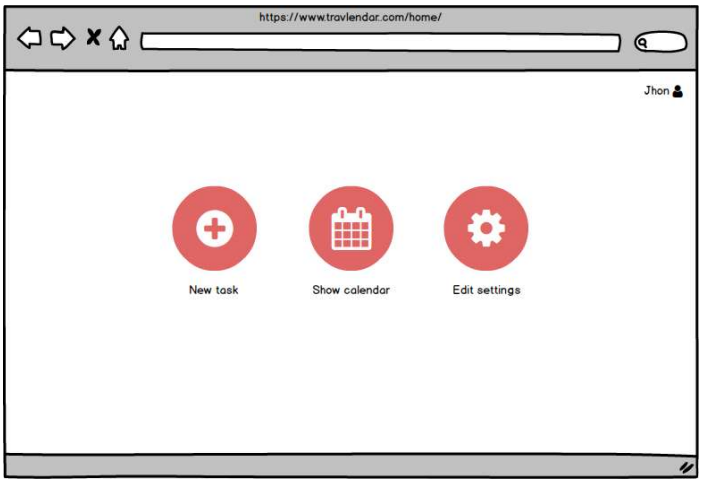


Preference views

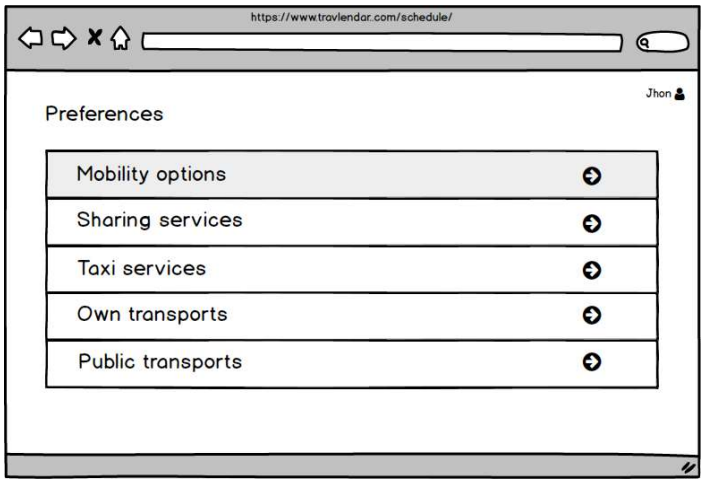


My passes views

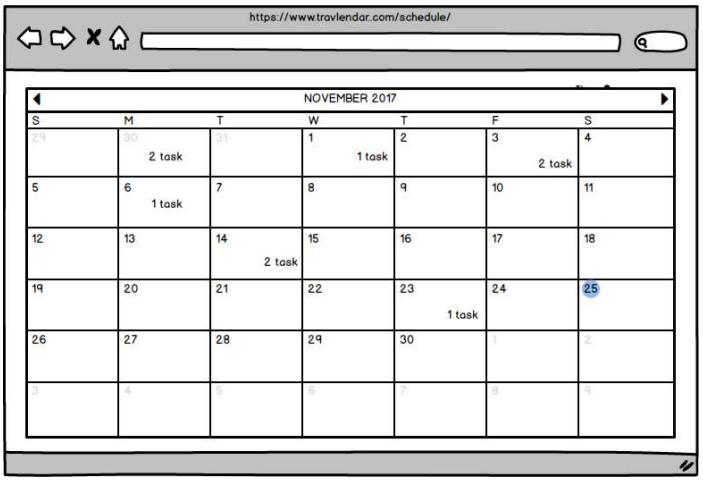
4.1.B. T+ Desktop



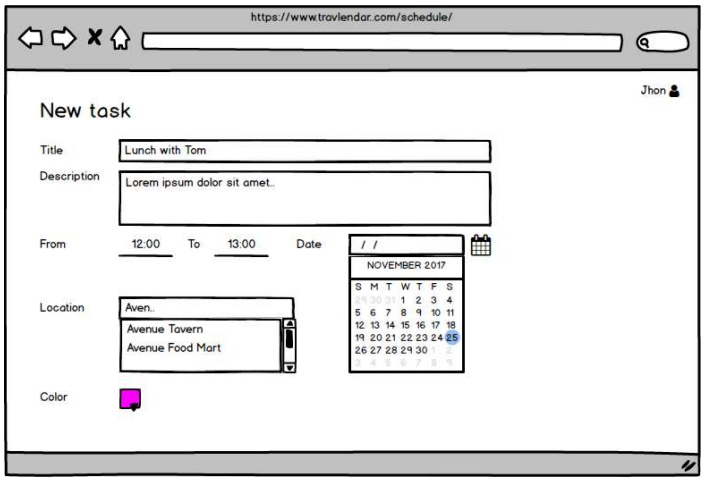
Main view



Preferences view



Schedule view

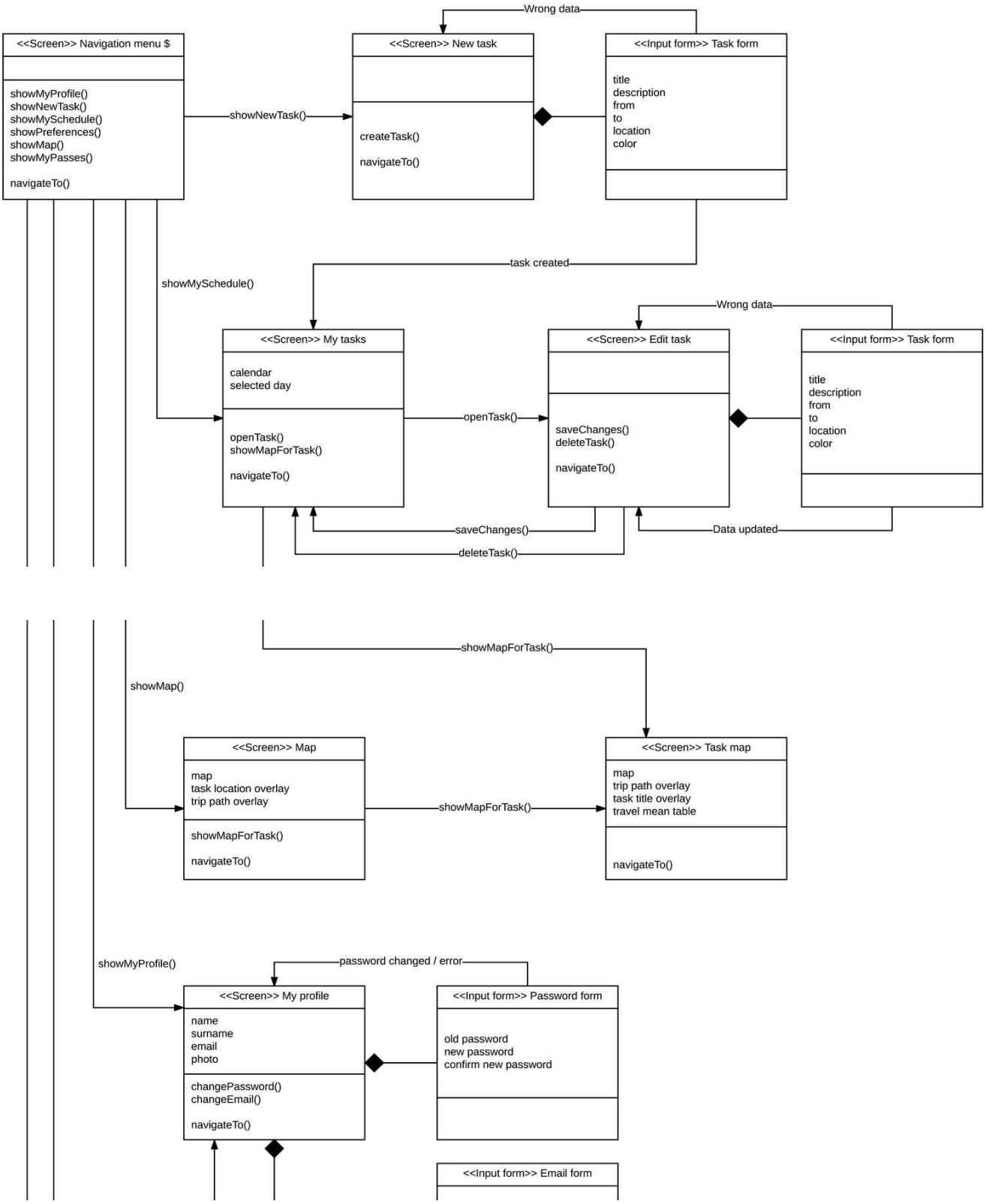


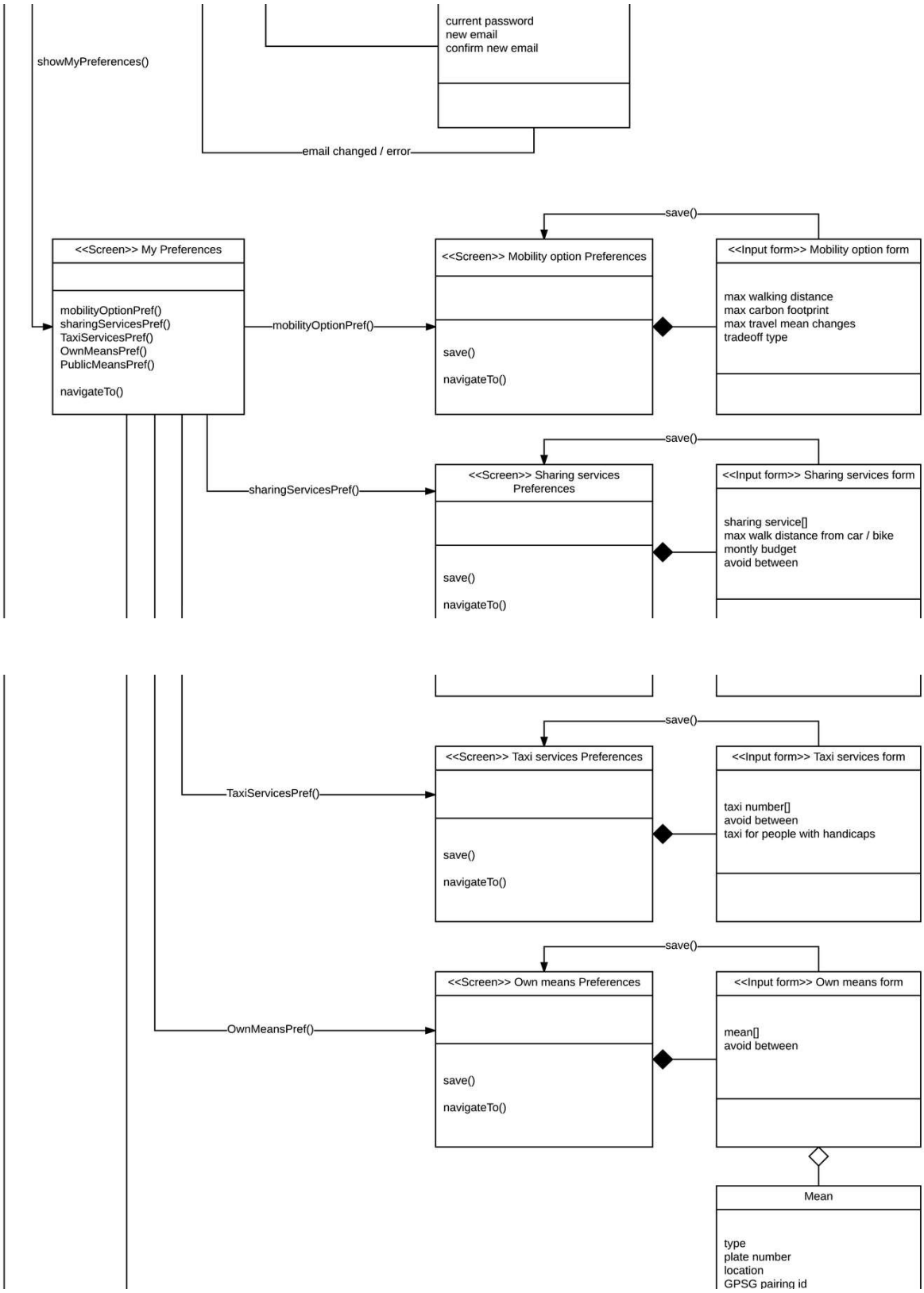
New task view

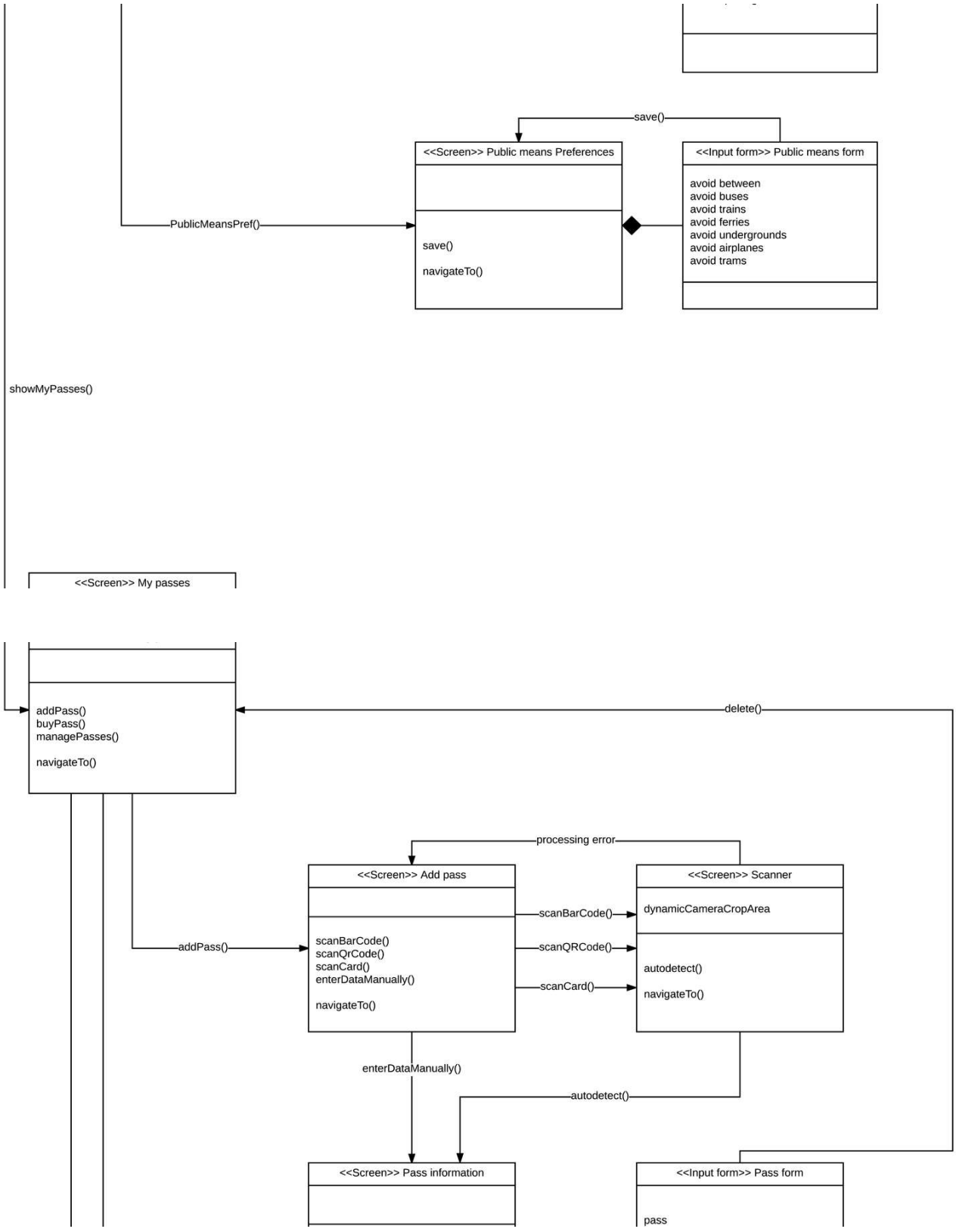
4.2. UX Diagram

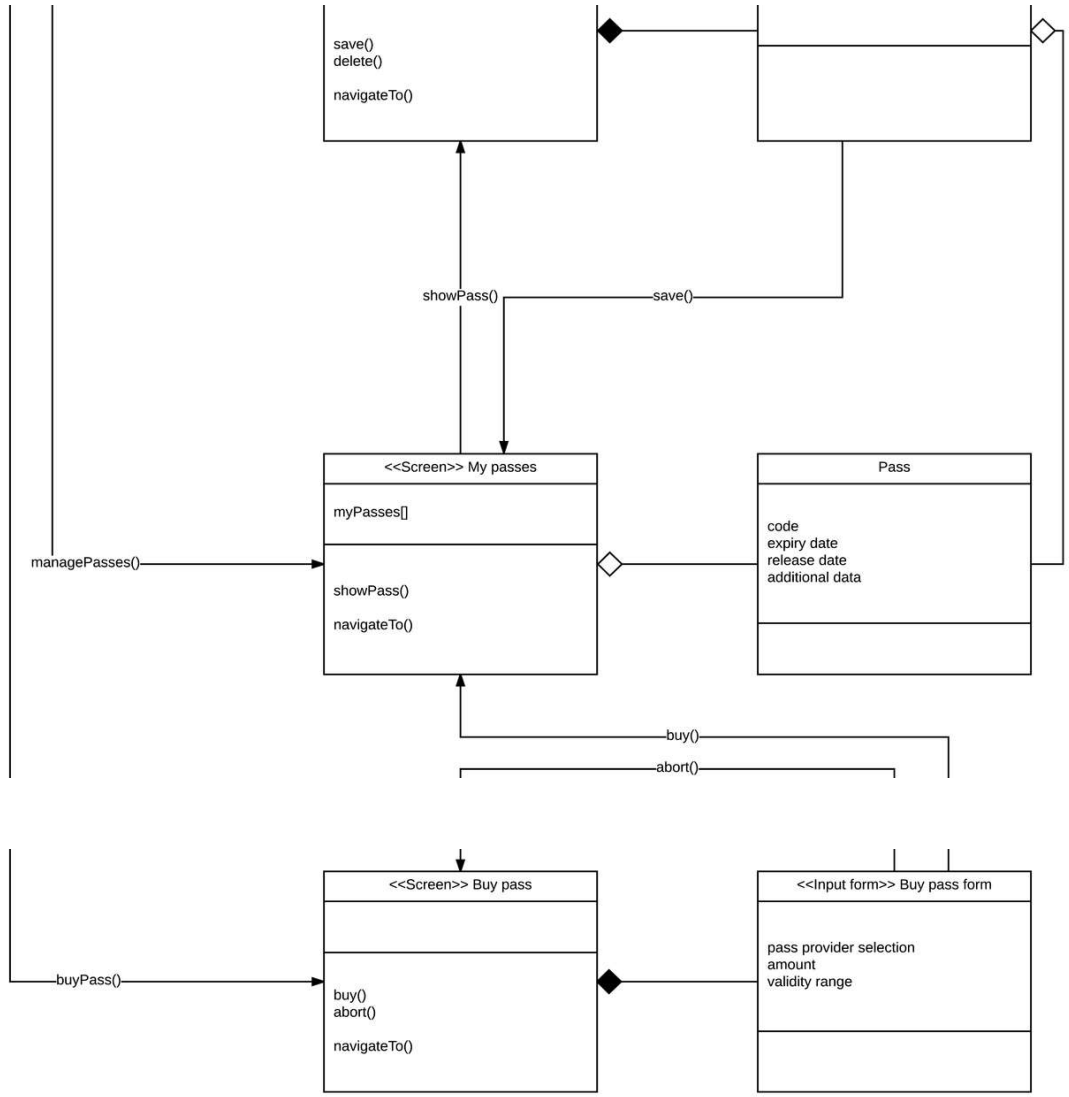
4.2.A. T+ Mobile

To provide a better overview of the interaction of the user with T+ Mobile a UX Diagram has been defined. Mockups for the most important screens have been defined in the RASD.

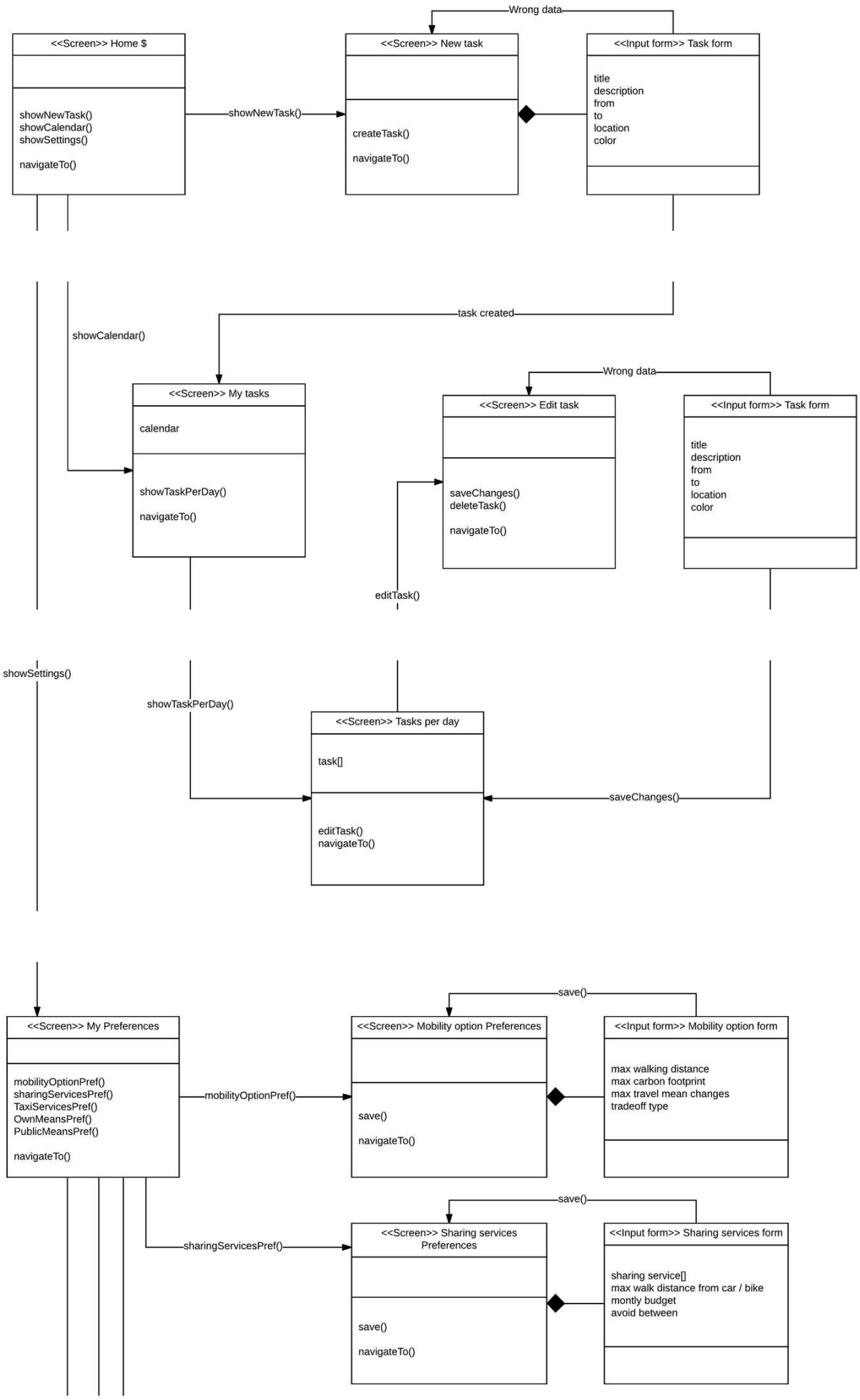

Wrong data
<<Screen>> Navigation menu $
showMyProfile()
showNewTask()
showMySchedule()
showPreferences()
showMap()
showMyPasses()
navigateTo()
showNewTask()
<<Screen>> New task
createTask()
navigateTo()
<<Input form>> Task form
title
description
from
to
location
color
task created
showMySchedule()
Wrong data
<<Screen>> My tasks
calendar
selected day
openTask()
showMapForTask()
navigateTo()
openTask()
<<Screen>> Edit task
saveChanges()
deleteTask()
navigateTo()
<<Input form>> Task form
title
description
from
to
location
color
saveChanges()
Data updated
deleteTask()
showMapForTask()
showMap()
<<Screen>> Map
map
task location overlay
trip path overlay
showMapForTask()
navigateTo()
showMapForTask()
<<Screen>> Task map
map
trip path overlay
task title overlay
travel mean table
navigateTo()
showMyProfile()
password changed / error
<<Screen>> My profile
name
surname
email
photo
changePassword()
changeEmail()
navigateTo()
<<Input form>> Password form
old password
new password
confirm new password
<<Input form>> Email form

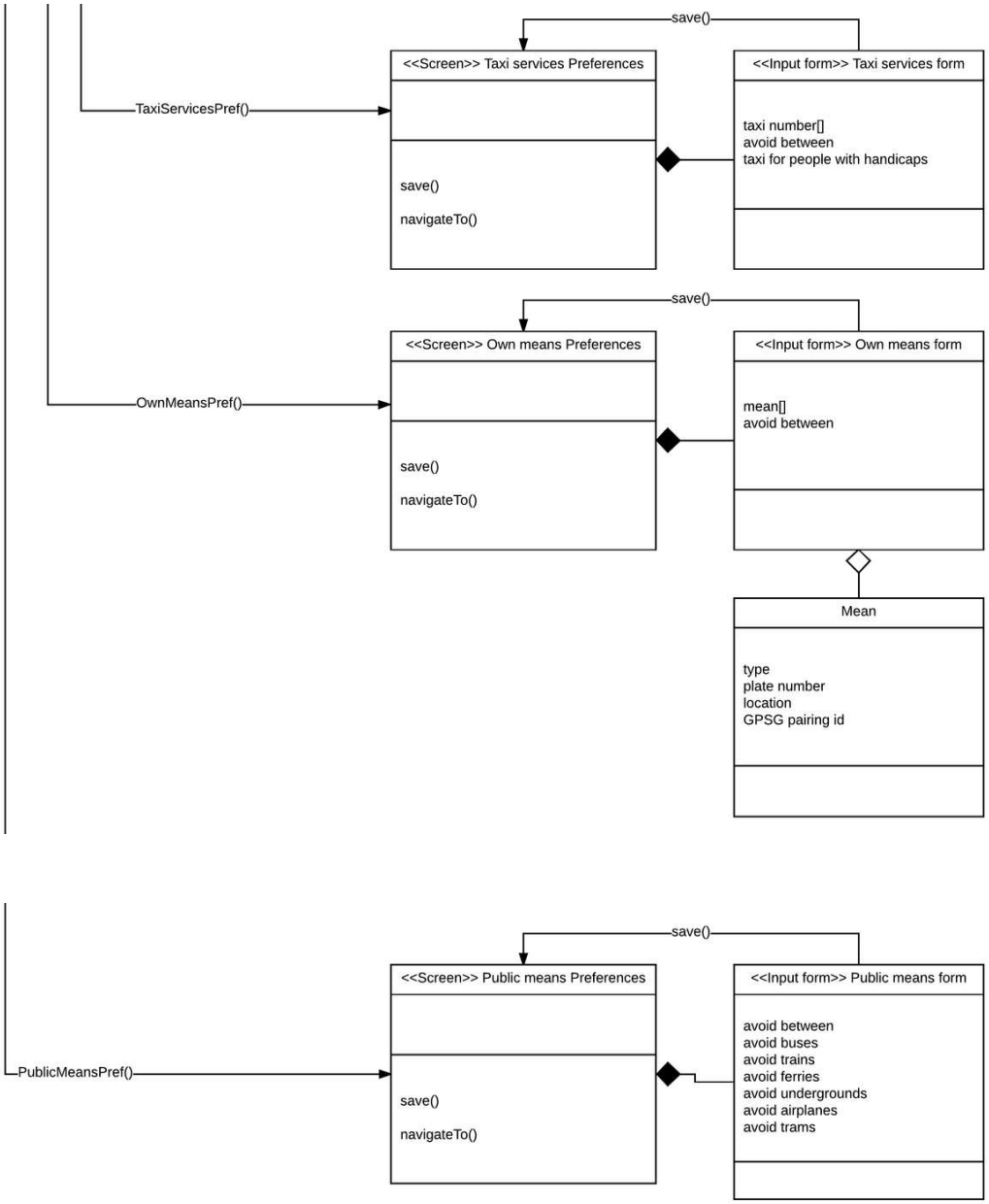

current password
new email
confirm new email
showMyPreferences()
email changed / error
save()
<<Screen>> My Preferences
mobilityOptionPref()
sharingServicesPref()
TaxiServicesPref()
OwnMeansPref()
PublicMeansPref()
navigateTo()
mobilityOptionPref()
<<Screen>> Mobility option Preferences
save()
navigateTo()
<<Input form>> Mobility option form
max walking distance
max carbon footprint
max travel mean changes
tradeoff type
sharingServicesPref()
<<Screen>> Sharing services Preferences
<<Input form>> Sharing services form
sharing service[]
max walk distance from car / bike
montly budget
avoid between
TaxiServicesPref()
<<Screen>> Taxi services Preferences
<<Input form>> Taxi services form
taxi number[]
avoid between
taxi for people with handicaps
OwnMeansPref()
<<Screen>> Own means Preferences
<<Input form>> Own means form
mean[]
avoid between
Mean
type
plate number
location
GPSG pairing id


save()
<<Screen>> Public means Preferences
<<Input form>> Public means form
avoid between
avoid buses
avoid trains
avoid ferries
avoid undergrounds
avoid airplanes
avoid trams
PublicMeansPref()
save()
navigateTo()
showMyPasses()
<<Screen>> My passes
addPass()
buyPass()
managePasses()
navigateTo()
delete()
processing error
<<Screen>> Add pass
<<Screen>> Scanner
dynamicCameraCropArea
scanBarCode()
scanQRCode()
scanCard()
addPass()
scanBarCode()
scanQrCode()
scanCard()
enterDataManually()
navigateTo()
autodetect()
navigateTo()
enterDataManually()
autodetect()
<<Screen>> Pass information
<<Input form>> Pass form
pass



4.2.B. T+ Desktop


Wrong data
<<Screen>> Home $
showNewTask()
showCalendar()
showSettings()
navigateTo()
showNewTask()
<<Screen>> New task
createTask()
navigateTo()
<<Input form>> Task form
title
description
from
to
location
color
task created
showCalendar()
<<Screen>> My tasks
calendar
showTaskPerDay()
navigateTo()
Wrong data
<<Screen>> Edit task
saveChanges()
deleteTask()
navigateTo()
<<Input form>> Task form
title
description
from
to
location
color
editTask()
showSettings()
showTaskPerDay()
<<Screen>> Tasks per day
task[]
editTask()
navigateTo()
saveChanges()
save()
<<Screen>> My Preferences
mobilityOptionPref()
sharingServicesPref()
TaxiServicesPref()
OwnMeansPref()
PublicMeansPref()
navigateTo()
mobilityOptionPref()
<<Screen>> Mobility option Preferences
save()
navigateTo()
<<Input form>> Mobility option form
max walking distance
max carbon footprint
max travel mean changes
tradeoff type
save()
sharingServicesPref()
<<Screen>> Sharing services Preferences
save()
navigateTo()
<<Input form>> Sharing services form
sharing service[]
max walk distance from car / bike
montly budget
avoid between



5. REQUIREMENTS TRACEABILITY

Functional Requirements

In this section we will specify how the components described in the DD can meet the goals and requirements identified in the RASD.

- [G1.1] Allow every registered user to submit new tasks
  - **Account manager** that interacts with account database in order to store the task in the local database
  - **Account Data Manager** that interacts with users database in order to store the task also in the tier 3 database.

- [G1.2] When unattendable tasks are submitted (with time overlaps or in any way that makes the attendance of all tasks physically impossible for transportation limitations) a warning is created
  - **UI manager,** to interact with the user
  - **Optimizer / On-Request Optimizer,** which computes the BMO
  - **Transportation Data manager,** which provide the information about transportation means
  - **Account manager / Account Data manager,** which provides the informations about other user's task

- [G1.3] Automatically accounts for travel times between appointments to make sure that the user is never late to attend his tasks
  - **Optimizer,** which computes the BMO and constantly updates its performance estimation to check whether the user can be in time for all his appointments
  - **Account manager / Account Data manager,** which provides all user's tasks and constraints
  - **Transportation Data manager,** which provides information on all the means of transport we can use

- [G1.4] Give easy directions that guide the user to every appointment
  - **Optimizer**, which provides a feasible path step by step
  - **UI manager,** which implements a suitable UI to guide the user

- [G2.1] Support a multitude of travel means, including walking, biking (own or shared), public transportation (including taxis), driving (own or shared)
  - **Transportation Data manager,** which generalizes the structure of all available transportation means
  - **API-specific modules**, which embed etherogeneus data about different means of transportation into the unified graph structure of the system

- [G2.2] Allow users to provide reasonable constraints on different travel means
  - **Account manager / Account Data manager:** organizes and manages submitted constraints

- [G2.3] Allow users to specify preferences on proposed solutions (time-cost trade-offs, additional general constraints
  - **Account manager / Account Data manager:** organizes and manages submitted preferences

- [G3.1] Suggest the best combination of transportation means considering the preferences of the user
  - **Optimizer / On-Request Optimizer**: takes into account the given preferences in proposing a solution minimizing the specified trade-off without violating constraints

- [G3.2] Automatically adjust the proposed travel solutions and notify the change to the user every time an environmental event happens (such as weather conditions)
  - **Ephemeral Data Manager:** collects all last-minute data about the status of transportation means
  - **Client Feed Manager**: selectively forwards updates about the current status of transportation means to clients
  - **Optimizer**: upon finding a BMO constantly updates current data in order to have a real-time estimate of its performance
  - **UI manager:** notifies the user as soon as the proposed solution has been changed to adapt to environmental changes

- [G4] to allow users to register to T +, to access and manage their account
  - **Account manager / Account Data manager**, which stores the user's general information

- [G5.1] to allow users to declare a day /week/ season pass and take it into account for cost computations
  - **Account manager / Account Data manager,** which stores the user's passes
  - **Optimizer / On-Request Optimizer:** provides a solution taking into account submitted passes to better estimate its real cost for the user

- [G5.2] to allow users to buy transportation tickets and passes
  - **Payment Manager,** which interacts with the transportation APIs to buy tickets or passes
  - **Account manager / Account Data manager,** which store the user new pass in order to take care of it during BMO computation

- [G5.3] to allow users to locate and reserve (if possible) the car or bike of a sharing system which is most suitable to their appointments
  - **Transportation Data Manager,** which interacts with the sharing systems APIs to locate and reserve a car/bike

Non-Functional Requirements

In this section we want to specify how the components described in the DD can meet non functional requirements described in the RASD and in particular reliability, availability, security maintanability and portability.

The usage of a fat-client provide Travlendar+ a great level of reliability. In fact the possibility to submit new tasks and compute BMOs inside the home city is fully autonomous on the client application, and thus this main service is granted even in the case of server failure. Secondary services such as device synchronization and transportation data updates will count on a redundant architecture to grant at least a 99.9% reliability during the uptime. Finally, not relying directly on a single external service (such as google maps) but on many external services that provide the data needed (that are saved in the transportation database), T+ system can safely endure the fact that some of these external services do not work for some time.

Security is achieved through both system and hardware measures. Account informations are encrypted on our servers and are not sent back to grant privacy but only used for device synchronization. Payment card information is kept local on the device where it is submitted
(and thus it will not be supported by device synchronization) to make sure that even our server system never receives this information. Moreover, the designed architecture makes our system even safer since personal user data is only saved on the tier 3 server and its network layer interacts only with the application servers in order to protect data from external attacks.

Portability is achieved because the application will be available in iOS, Android and Windows Phone platforms but also on classical web browsers.

6. IMPLEMENTATION, INTEGRATION AND TEST PLAN

The implementation and testing of the T+ system will focus on anticipating the critical points of the architecture and their integration.
The most critical points are the following:

- The Transportation Data Manager component must provide a common abstraction to make all transportation information processes able to interact undirectly. It should also optimize the access to transportation data, thus being a central point for both integration of various components, performance of the BMO computations and memory consumption of both T+Mobile and Tier2 Servers.
- The Optimizer module is the one performing all heavy computations, thus it is a key-point to test for performance requirements together with the implementation of the Transportation Data Manager.

Also two main almost independent areas can be identified to be developed and tested in parallel:

- The account managing modules in both client and server T+ systems interact with each other providing an autonomous set of functionalities including personal information management, calendar management, and

synchronization without particular performance requirements.
- The transportation managing modules collect, integrate, share and elaborate transportation data to provide to the end-user the BMO according to its preferences. The interaction of this block of components with the other one is limited to the Account Information Interface between the optimizers and the structured account managers, and it can be easily replaced by a stub to completely isolate the two blocks.

Few modules are left out of the previous separation:
- The network modules are low level components shared between the two main blocks for network communication. These modules are highly coupled with both blocks, but they show neatly distinguishable features to serve either one or the other block. These features can be developed independently.
- The GUI is a high level component loosely coupled with both blocks, thus it can be developed and maintained independently with simple stubs for the Plan Interface, the Account Management Interface and the Payments Interface.
- The Payments component embeds in the application all payment methods that the user could need with loose coupling with the GUI. The specific network features it needs should be developed first and independently from all other network features and modules, then it can be implemented and tested with a very simple driver simulating the requests coming from the GUI.

Transportation Managing Plan

The transportation managing block includes the set of services associated with transportation data, and their common back-end. It includes both of the critical points highlighted above, so it must be treated with special care.
This block includes:
- The Transportation Data Manager
- The Optimizer and On-Request Optimizer
- The Ephemeral Data Manager
- The Client Feeds component
- All API-specific modules

The core of its back-end, the Transportation Data Manager, is foundamental for both integration and performance of the entire block, so the approach on developing this part will be bottom-up.

Transportation Data Manager Plan
This module is the core of the transportation services, thus it must be implemented first. It can be based on a COTS database service, and its submodules should be implemented and unit tested independently. They should be then integrated in the whole component and tested for both functionality and indicative performance using simple stubs to provide random updates and different drivers to simulate different loads. Its parameters can be tuned to achieve the best trade-off between memory consumption and CPU usage for serialization and deserialization, but their value should not be considered final until it is integrated with the Optimizer and some basic API modules to tune them with real data.

Ephemeral Data and Client Feeds Plan
These components are relatively simple index and lists managers, thus they can be easily developed in parallel with the Transportation Data Manager without the need of intensive testing and tuning: they should pass unit tests and they should just provide and register data as fast as possible. They are expected to take far less effort than the Transportation Data Manager.

Optimizers Plan
The server side On-Request Optimizer is a light version of the client side Optimizer without the service of continuous performance monitoring of the current BMO (as servers will not be dedicated to single BMOs). For this reason the full Optimizer can be developed starting from the On-Request Optimizer adding functionality in terms of the additional

submodules needed. These modules should be built after the Transportation Data Manager is fully functional (still untuned) and they can be tested using pseudorandom transportation data from the stubs of the Transportation Data Manager and some collections of various combinations of reasonable constraints and preferences from a simple stub of the Account Information Interface.
Their performance is critical for T+, thus they should be tested and integrated directly with the Transportation Data Manager to monitor local performance bottlenecks and provide a finer parameter tuning. Much

API modules Plan
API modules should provide real-world data to our system. Their structure should be focused more on simplicity than optimization, as being able to build new API modules fast and reliably is the core of T+ coverage expansion, while integrating the newest data from outside into our system is not performance-critical (delays in the order of hours is often tolerated, extreme cases tolerate delays of tens of minutes). The basic modules should be developed first, providing information about the topology of covered cities and walk/bike/car speed and availability on each street.
In both the development phase and during later maintenance they should be tested on instances of the Transportation Data Manager backed by a copy of the current transportation database, in order to be sure not to introduce incorrect data into the system.
An integration test should follow the success of unit tests on the first modules in order to tune the Transportation Data Manager parameters about aggregation of different coordinate locations into one graph node.

Transportation Network Features Plan
Network modules offer easily distinguishable features between Transportation and Account blocks. As the Transportation block follows a bottom-up development and testing policy, network features about transportation should be developed first. Their implementation and testing is not considered to be time consuming, for this reason it should be placed along a non-critical path such as after the Ephemeral Data and Client Feeds Plan, but in principle they could be built and tested independently.

Final Transportation Integration
As soon as all the components listed above are ready and they have been integrated with their back-end, an integration test should be performed with the whole transportation block.
This integration test is not expected to find important integration issues as the main direct interactions between components have already been tested. After covering all interfaces in the block, this final integration has the advantage of providing to the transportation system real world data from the active API modules and real optimization tasks to finally tune the Transportation Data Manager and the Optimizer parameters. The same Account Information Interface stub used for the Optimizer modules testing can be used now to test the whole block on real cities. The time consumption of every activity should be monitored as well as graph data request patterns and database and network accesses and delays in order to find bottlenecks in the entire process of finding a BMO and keeping it up-to-date with different usage settings.

Account Managing Plan

The Account Managing block provides the features of storing and synchronizing user data across all its registered devices. The main focus on this block is security and reliability, while performance is secondary.
This includes:

- Client side Account Manager
- Tier2 Server side Account Data Manager
- Tier3 Server side Account Data Manager

These feature-oriented components allow us to easily build a driver simulating the GUI requests and stubs simulating lower level components interaction to test each feature independently.
This also suggests a top-down integration plan, as building lower level stubs is more manageable than simulating higher level drivers (except for the highest one, the GUI) in this case.

Client side Account Manager
The client side Account Manager should interact with the local DBMS, store and structure data for outside requests and notifications. It is a simple module that can be implemented and unit tested with little effort. Unit tests on this module will verify all basic requirements about account management (submission of new tasks and preferences).

Tier2 Server side Account Data Manager
The Account Data Manager on Tier2 should certify user identities and forward requests between the architectural layers. As with the client side Account Manager, this module is simple and requires very little effort for implementation and unit testing.

Tier3 Server side Account Data Manager
This component is the back-end for the synchronization service, for this reason it is more complex than the others of the same block. As this block is not expected to be in the critical path, it can be developed and tested after all client and tier2 server Account Managers are ready for the Account block final integration test.

Account Network Features Plan
Network modules offer easily distinguishable features between Transportation and Account blocks. As the Account managing block follows a top-down integration approach, account features in Network modules can be built and tested in sequence between the other three components.

Final Account Integration
As soon as upper modules are ready and unit tested, they can be integrated with a top-down approach following the order:
1. Client side Account Manager
2. Client side Network Manager - Account features
3. T2 Server side Client Network Manager - Account features
4. T2 Server side Account Data Manager
5. T3 Server side Secure Network Manager
6. T3 Server side Account Data Manager

The last integration test up to point 6 will be able to directly test multi-device synchronization and will monitor the security and robustness of the whole account management system.

Final System Integration and Test Plan

After all components are ready and all partial integration tests described above have been completed successfully, the system is ready for the final integration between GUI, Payments Manager, Account Managing block and Transportation block.
This final integration will mainly focus on the correct interaction between the GUI and the three other blocks (Plan Interface, Account Management Interface, Payments Interface) and between the Transportation and Account Managing blocks (Account Information Interface), which are the only interfaces left from previous integrations.

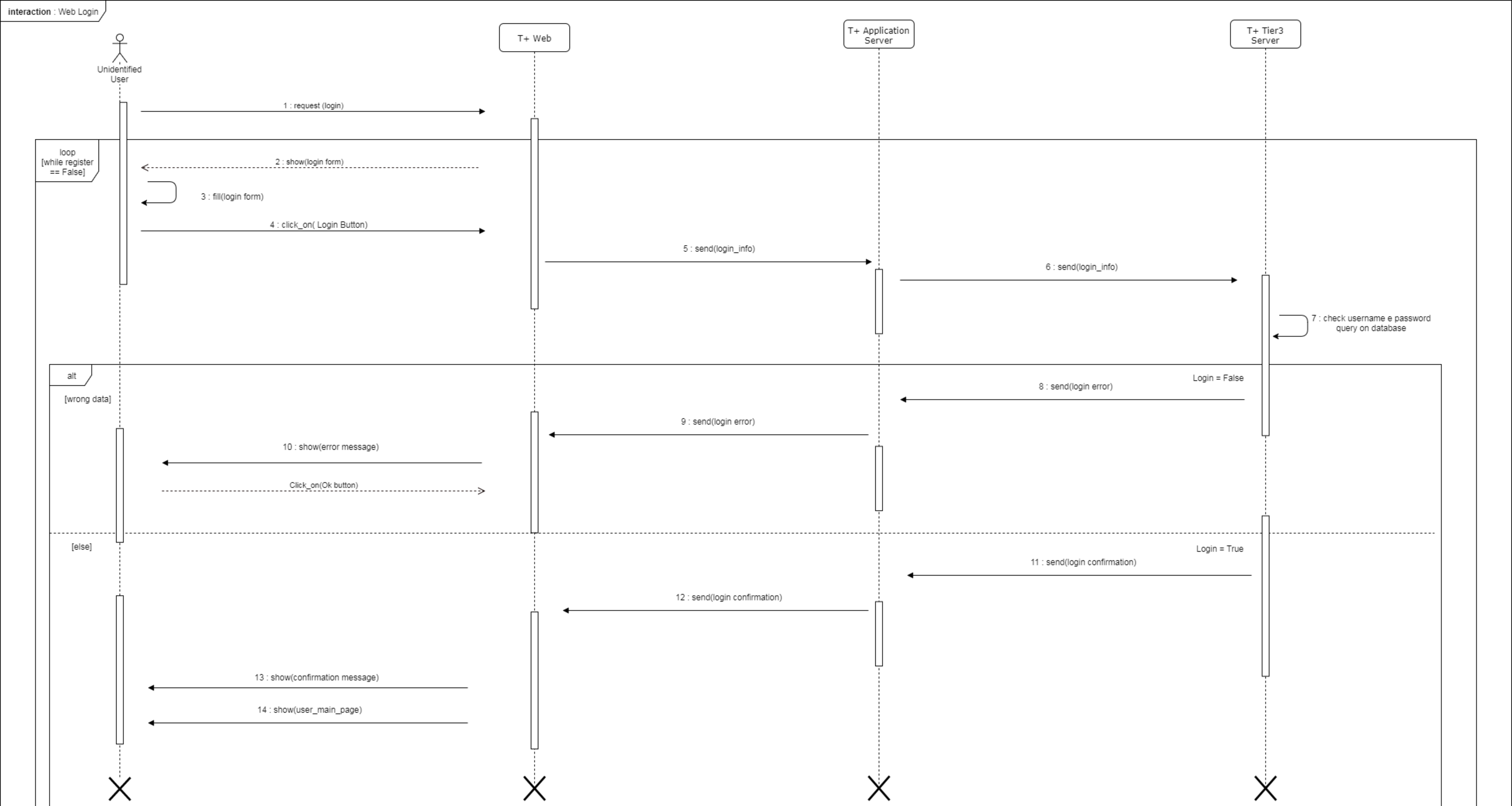
7. EFFORT SPENT

Lorenzo Barcella	17.25
Alberto Bellini	15
Luca Cavalli	19

8. REFERENCES

Reference Documents

- [1] Software Engineering 2 course slides.
- [2] Travlendar+ assignment.
- [3] The PostgreSQL Global Development Group
- [4] Nodejs Foundation


interaction : Web Login
Unidentified User
T+ Web
T+ Application Server
T+ Tier3 Server
1 : request (login)
loop [while register == False]
2 : show(login form)
3 : fill(login form)
4 : click_on(Login Button)
5 : send(login_info)
6 : send(login_info)
7 : check username e password query on database
alt
[wrong data]
Login = False
8 : send(login error)
9 : send(login error)
10 : show(error message)
Click_on(Ok button)
[else]
Login = True
11 : send(login confirmation)
12 : send(login confirmation)
13 : show(confirmation message)
14 : show(user_main_page)

interaction : Mobile Login

Unidentified User

T+ Mobile UI Manager

T+ Mobile Account Manager

DBMS

1 : request (login)

loop [while login== False]

2 : show(login form)

3 : fill(login form)

4 : click_on(Login Button)

5 : send(login information)

6: select(...)

return

7 : check username and password

alt

[wrong username or password]

8 : send (login error) Login = False

9 : show (error message)

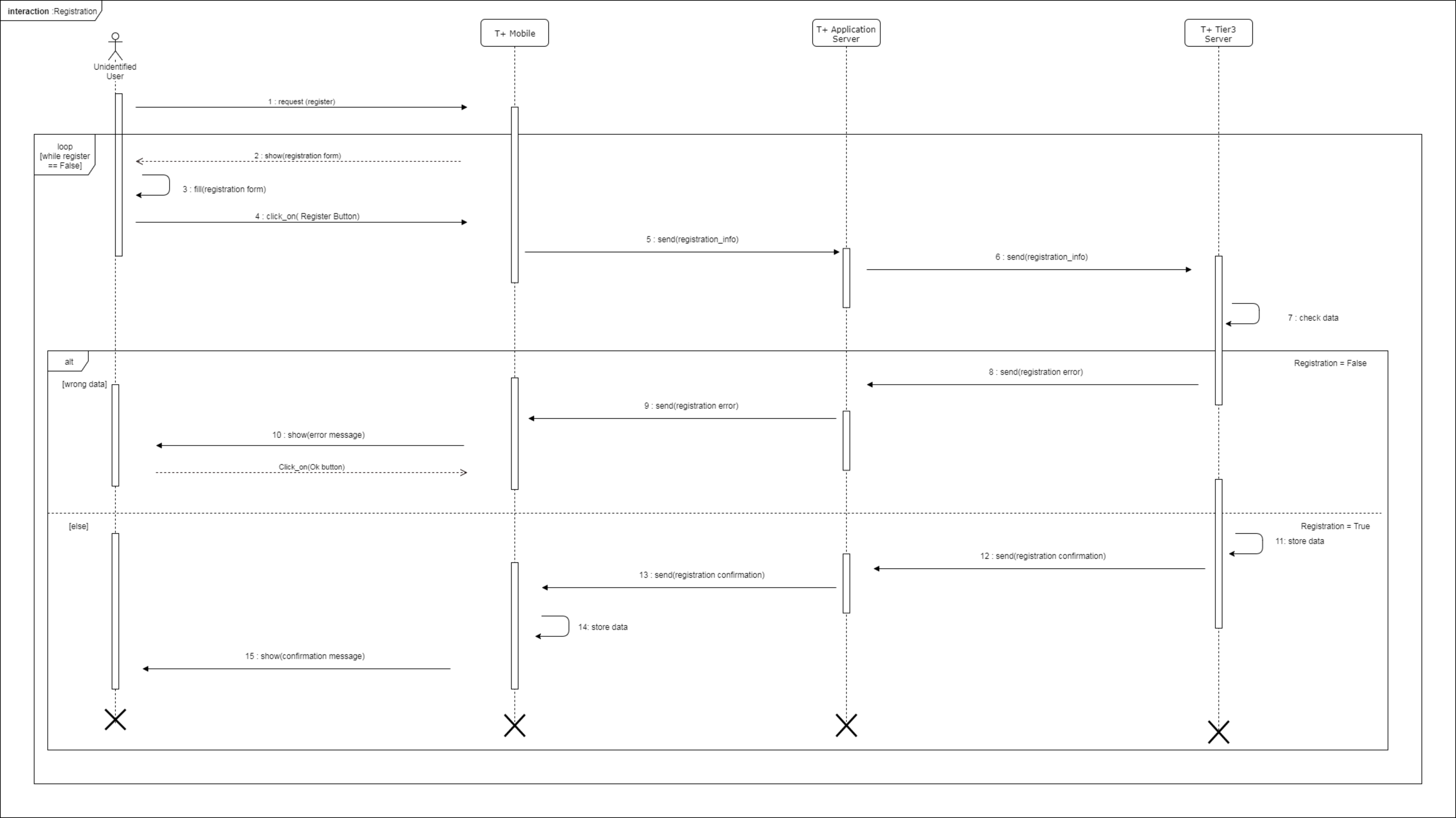
click_on (OK_button)

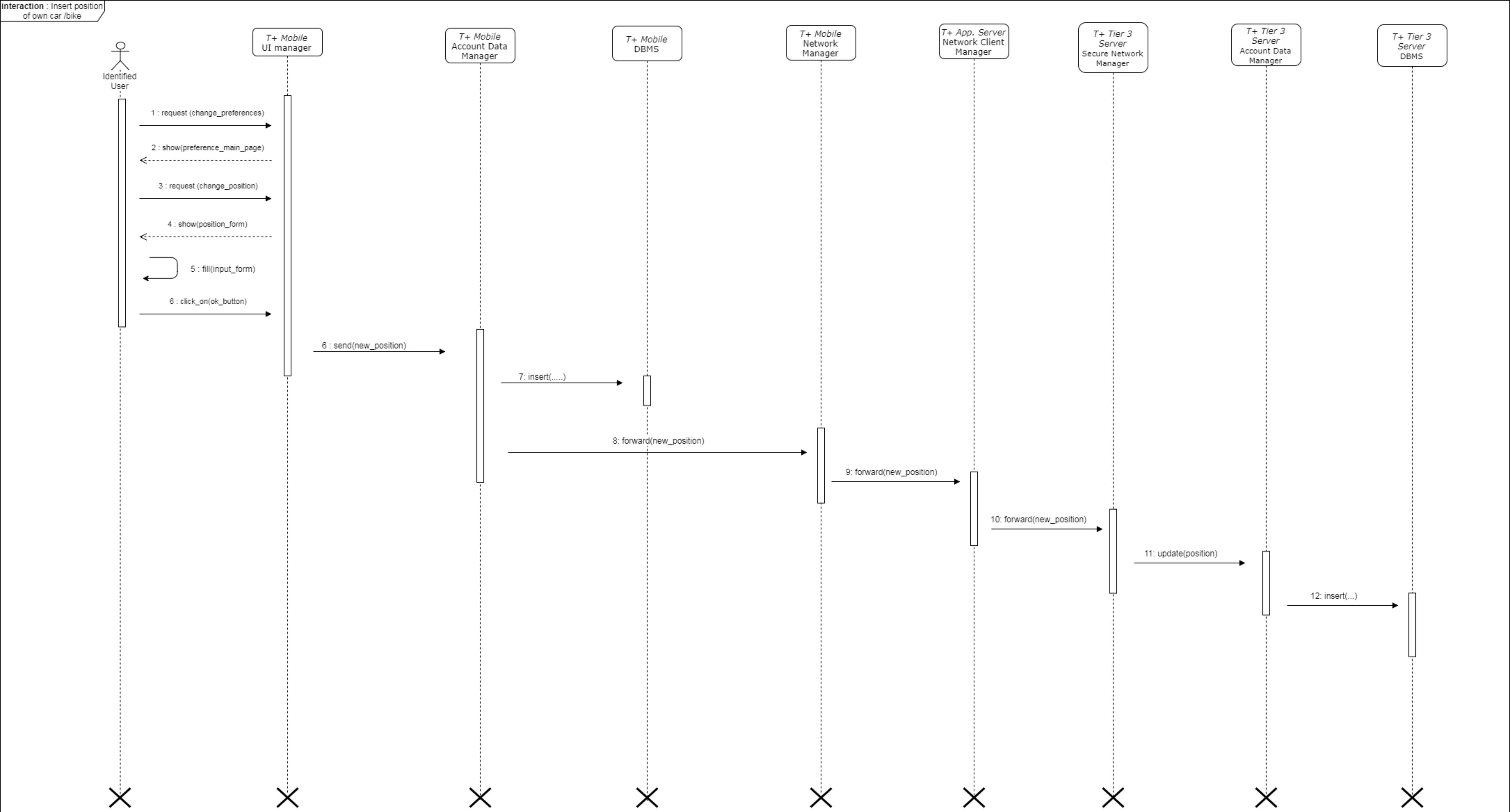
[else]

10 : send (login confirmation) Login = True

11 : show (confirmation message)

12 : show (user_main_page)


interaction :Registration
Unidentified User
T+ Mobile
T+ Application Server
T+ Tier3 Server
1 : request (register)
loop
[while register == False]
2 : show(registration form)
3 : fill(registration form)
4 : click_on(Register Button)
5 : send(registration_info)
6 : send(registration_info)
7 : check data
alt
[wrong data]
8 : send(registration error)
Registration = False
9 : send(registration error)
10 : show(error message)
Click_on(Ok button)
[else]
Registration = True
11: store data
12 : send(registration confirmation)
13 : send(registration confirmation)
14: store data
15 : show(confirmation message)



After the update of the position the system update the BMO. We omit this passage because is well descripted in the New Task Creation sequence diagram

interaction : New Task Creation

Identified User

T+ Mobile UI manager

T+ Mobile Account Data Manager

T+Mobile Optimizer

T+ Mobile Transportation Data Manager

T+ Mobile DBMS

T+ Mobile Network Manager

T+ App. Server Network Data Manager

T+ App. Server Clients Feed Manager

T+ App. Server Ephimeral Data Manager

T+ Tier 3 Server

1 : request (create_new_task)

2 : show(new task input form)

3 : fill(input form)

4 : click_on(Create Task)

5 : send(new_task_info)

6: insert(.....)

7: forward(task_info)

8: forward(task_info)

9 : synchronization

10 : send(new_task_info)

11 : send(account_info)

12 : ask_for(transportation_info)*

13: ask_for(ephimeral_info) [optional]

14: send_request

15: ask_for(eph_info)

16: ask_for(eph_info,client)

17 : ephimeral_data_info

18 : ephimeral_data_info

19 : Response

20: ephimeral info [optional]

21: transportation_info

22 : BMO computation

23: send(BMO information)

24: show(confirmation message)

25: show_new_BMO