

SMART and FLEXible mobile DATA COLLECTOR for GIS

(Acronym, MOBILO)

ENTERPRISES 0916/0055

[D20] Report on Prototype's Architecture

Deliverable n.	D20	Deliverable title	Report on prototype's architecture
Workpackage	WP8	WP title	System integration & testing
Editors	Elias Frentzos (GEO), Dimitrios Skarlatos (CUT)		
Contributors	Elias Frentzos(GEO), Dimitrios Skarlatos (CUT), Maria Aristodimou (PWD), Petros Katsikadacos (GEO), Marinos Vlachos (CUT)		
Status	Final		
Distribution	Public		
Issue date	2021-03-31	Creation date	2021-03-01

Contents

LIST OF FIGURES	4
LIST OF TABLES	5
LIST OF ABBREVIATIONS.....	6
REVISION CHART AND HISTORY LOG.....	7
1 Introduction.....	9
2 MOBILO Prototype	10
2.1 Prototype System Architecture	12
2.2 Prototype design	14
2.3 Limitations	15
2.4 Low-cost System.....	16
3 Source Code Management.....	16
3.1 Version Control Software (VCS).....	17
3.1.1 VCS basic functions.....	17
3.2 Basic version control procedures	18
3.2.1 Example based on a typical scenario.....	19
3.3 Types of Version Control	19
3.3.1 Centralized Version Control System (CVCS)	19
3.3.2 Distributed Version Control Systems (DVCS).....	20
3.4 Git Branching Models	21
3.5 GitFlow Workflow Description	22
3.5.1 GitFlow Branches.....	22
4 Integration Results – system’s accuracy.....	23
4.1 Discussion	24
4.2 Experimental Setup	27
4.3 Results	28
5 Licensing	29
5.1 General Description.....	30
5.2 LockerService Project	31
5.3 LiveLockerClient Project	33
5.3.1 User Interface	35
5.3.2 API Usage.....	35
5.4 Customer Manager Project	36
5.4.1 UI classes	36

6	Conclusions.....	37
7	References.....	39

LIST OF FIGURES

Figure 1: Preliminary design	10
Figure 2: First working version	10
Figure 3: Initial carrier plan	11
Figure 4: Implemented carrier plan.....	11
Figure 5: Semi-final carrier	12
Figure 6: MOBILO Carrier (Prototype) architecture	13
Figure 7: MOBILO Carrier Prototype design.....	14
Figure 8: All the components of the system arranged in the carrier	15
Figure 9: MOBILO carrier mounted on top of a vehicle	15
Figure 10: Low-cost system on top of a vehicle	16
Figure 11: GitFlow example diagram.....	23
Figure 12: Geodetic vs photogrammetric reference systems	25
Figure 13: Synchronization test among the cameras, with the GPIO cable	27
Figure 14: Control points in the test field.....	27
Figure 15: Test field with MOBILO trajectories	28
Figure 16: LiveLocker system architecture.....	30
Figure 17: Sample Code.....	34
Figure 18: Locking algorithm with offline check.....	34
Figure 19: FrmLocker UI	35
Figure 20: Customer Manager Main Form	36
Figure 21: Customer info dialog	37
Figure 22: DB connection	37

LIST OF TABLES

Table 1: Parts included in MOBILo system	13
Table 2: Accuracy of the stereo rig.....	25
Table 3: Positioning subsystem error transmission	26
Table 4: Positioning subsystem error transmission with heading calibration	26
Table 5: Interior and relative orientation results.	28
Table 6: Accuracy of solutions.....	29
Table 7: Repeatability of solutions.....	29

LIST OF ABBREVIATIONS

API	Application Programming Interface
GPS/GNSS	Global Positioning System / Global Navigation Satellite System
INS/IMU	inertial navigation system / inertial measurement unit
DDRM	Distributed Digital Rights Management
VCS	Version Control Software

REVISION CHART AND HISTORY LOG

REV	DATE	REASON
0.1	01/03/2021	Initial
0.2	15/03/2021	Draft
0.3	31/03/2021	Final

Executive Summary

This deliverable is part of the results of MOBILO's project WP8 and focuses on the System's architecture and several other integration issues. MOBILO general architecture consists of a Mobile Mapping System that includes several components and is able to be mounted on top of vehicle in order to collect georeferenced video data, as well as three software components which are used to calibrate the system, collect and process the actual data. Our final system architecture is based on a rigid Plexiglas box that hosts all parts in its interior, is attached to the carrying vehicle with magnetic mounts and provides connection with a single cable to the managing computer. Regarding the system's performance, in our experiments we establish that the accuracy of the system is below 0.50 m which outperforms our initial expectations. There are also two other issues that emerged during system integration: The employment of VCS enables the continuous development of our system, while the total refactoring of an existing licensing mechanism led to a complete DDRM (Distributed Digital Rights Management) system for software developers that enable remote access control to their propriety applications. As emerges by the integration, given that our system provides points with 0.5m uncertainty in absolute coordinates, it appears as a nice alternative to more expensive MMS for several applications.

1 Introduction

MOBILO general architecture consists of a Mobile Mapping System that includes several components and is able to be mounted on top of vehicle in order to collect georeferenced video data, as well as three software components which are used to calibrate the system, collect and process the actual data. In this deliverable, we focus on the task of MOBILO System Integration. We describe the evolution of the MOBILO System Carrier, preliminary designs and issues appeared, version control software management strategies that helps in software integration and continuous development and software licensing methods that are essential for commercial software exploitation.

More specifically, the issues addressed in this deliverable are as follows:

- **Mobilo Prototype:** During project's evolution we have developed several versions of the system's carrier, experimented with materials, parts, and designs. The final design is based on a rigid Plexiglas box hosts all parts in its interior, is attached to the carrying vehicle with magnetic mounts and provides connection with a single cable to the managing computer. Box closes with a lid that enables easy access to the system's parts.
- **Source Control Management:** During the task of project integration several revisions of the underlying software were required; hence, the need for a version control software emerges. We have therefore invested on source code management via version control software (Git) [1], that makes easy to work on large groups of developers producing quality code throughout the product's life cycle. Version Control also enables to have control over the code, track the changes made on each version and via GitFlow program multiple scheduled releases and provides a strong framework of roles and responsibilities to each development branch.
- **Integration results:** The results of the system's integration are measured in terms of the accuracy achieved by our system. In order to determine it, we first invest on a theoretical discussion regarding error origin and propagation, and then we proceed to experimental data gathering on a test field with premeasured targets. The results of our experiments establish that the accuracy of the system is less than 0.50 m providing therefore an extremely attractive low-cost alternative to far expensive existing Mobile Mapping Systems.
- **Licensing:** All developed software components are protected and locked with a licensing system that is based on a service running the cloud. The licensing mechanism locks the software over specific hardware properties of a workstation, providing at the same time the ability to transfer the license between workstations. The license check is performed over the web and on a database that stores client and software data. The developed system is a complete DDRM (Distributed Digital Rights Management) system for software developers that wishes to have remote access control to their propriety applications.

The rest of this document is structured as follows: Section 2 provides info about the Carrier design and implementation as well as the total system architecture. Section 3 focus on source code management issues arising when dealing with software projects developed and

maintained by large teams and the employment of Git Flow [2] in our procedures. Section 4 provides the results of the total implementation in terms of accuracy achieved. Section 5 describes the licensing mechanism integrated into our software component and Section 6 provides conclusions and possible future work.

2 MOBILO Prototype

During project's evolution we have developed several versions of the system, experimenting with materials, parts, and designs. We used preliminary designs with easy-to-handle materials as the one of Figure 1. The first working version on which all parts were connected to a laptop computer hosting the collection software can be found in Figure 2.



Figure 1: Preliminary design

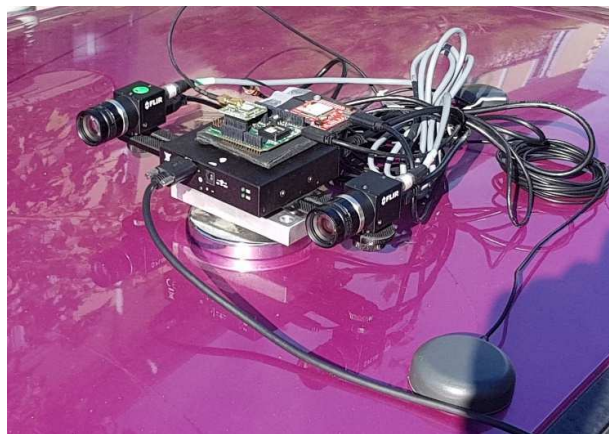


Figure 2: First working version

These preliminary designs had several drawbacks such as small base length etc. Therefore, In the next steps we started to produce designs of system carriers as the one of Figure 3 which is implement in Figure 4. This is a design which was implemented with plexiglass, however it features rather small base length, while it was proven to be less rigid than it should be. All carriers implemented were attached on top of the carrying vehicle with appropriate magnetic mounts.







Figure 5 displays the semi-final carrier which addresses all these issues and was used for testing purposes. In this design some parts are still exposed to environment conditions (e.g., cameras, USB hub), while the need for an external power supplier through the car's battery still existed.



Figure 5: Semi-final carrier

2.1 Prototype System Architecture

The final parts included in our system have extensively presented in several deliverables and are summarized in Table 1.

Part	Name
	Blackfly S USB3 . BFS-U3-50S5C (2x)
	Wide-Angle "FUJINON HF6XA-5M" Machine Vision Lens (2x)
	SparkFun GPS-RTK2 Board - ZED-F9P
	XSens MTi-7 DK
	Tallysman TW-150
	Tallysman TW-7872 (L1/L2/L5)





Part	Name
	Magnet mounts (2x)
	USB HUB with
	Powerbank
	Several cables (GPIO, SMA, USB)

Table 1: Parts included in MOBILO system

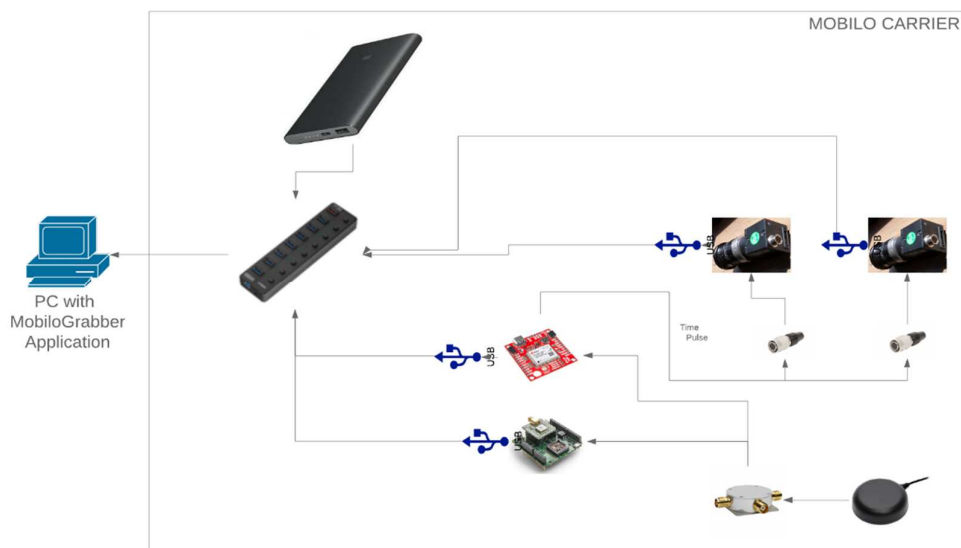


Figure 6: MOBILO Carrier (Prototype) architecture

All parts are connected to each other as described in Figure 6. All parts are placed inside the MOBILO carrier. The two cameras, GPS/GNSS board and INS / IMU boards are connected via USB cables to a USB Hub which is powered by an appropriate power-bank 25000 mAh capable for providing power for at least 8 hours of operation. Cameras with the appropriate lenses are connected to the time pulse interface of the GPS / GNSS board via GPIO cables. Finally, GPS / GNSS and INS / IMU boards are connected via SMA cables to a GPS signal splitter (TW150) which splits the signal acquired from a single L1 / L2 / L5 antenna

(TW7872). Finally, the hub is connected to a laptop on which MobiloGrabber software operates which is responsible for data collection in the field.

2.2 Prototype design

For the final MOBILO Carrier we used a box which was especially designed to fit to our needs. The box was made by plexiglass 7 mm thick providing a highly rigidly mount. The box is able to open on its top so as to provide access to all parts. This is due to the need for charging the power bank, opening, and closing power buttons of the USB hub etc. On the lid of the box (top side), there are three pre calibrated positions on which the GPS antenna can be placed. Cameras are mounted on the box though four screws for each camera so as to ensure that they stay in a stable position. At the bottom of the box there are two positions for fitting the magnet mounts that are used to attach the carrier to a vehicle. The final box design can be found in Figure 7.

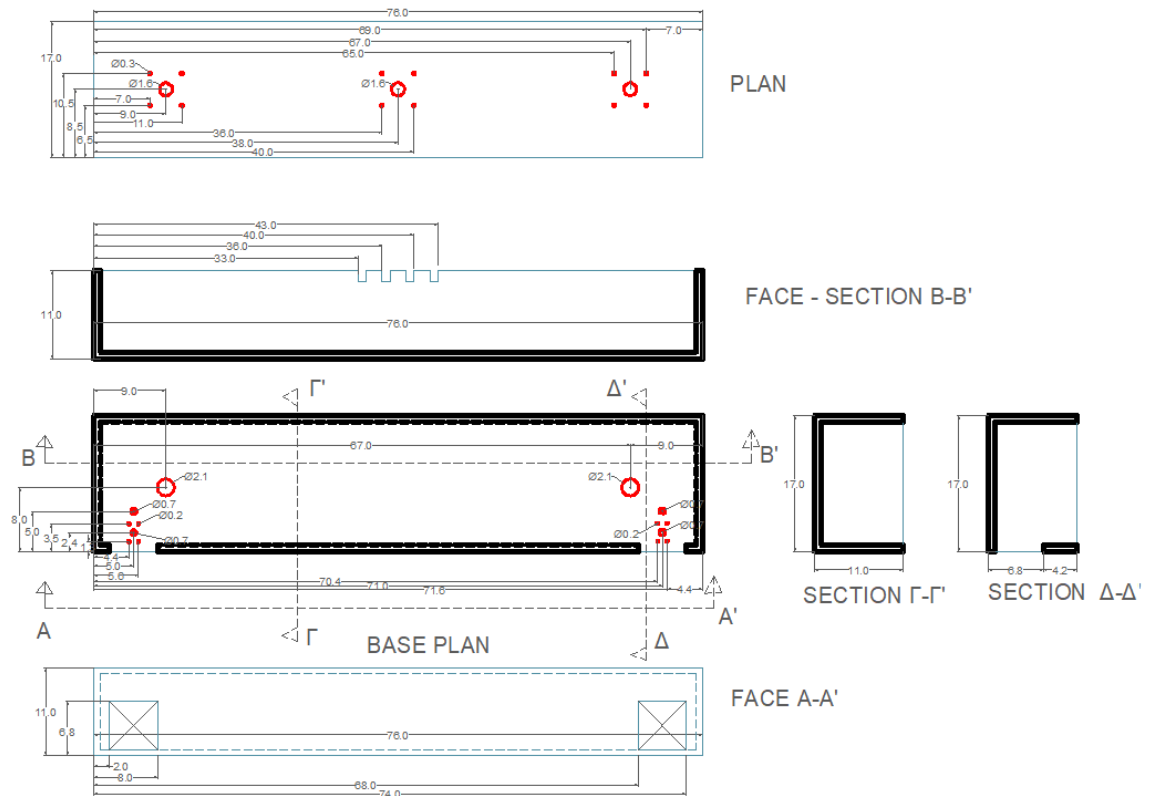
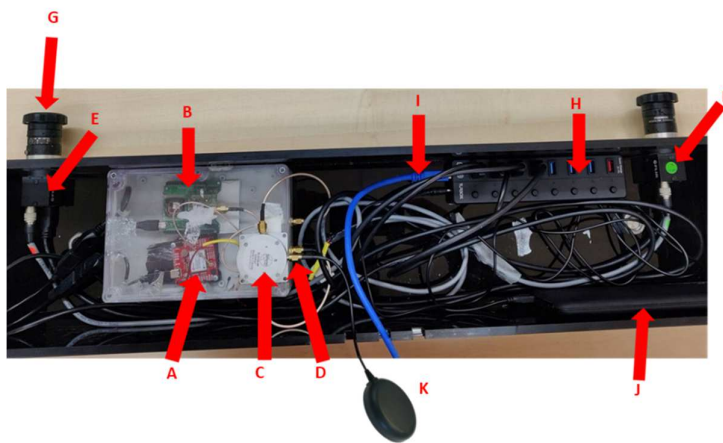


Figure 7: MOBILO Carrier Prototype design



- A. GPS / GNSS ZED-F9P module.
- B. INS Xsens MTi-7.
- C. Antenna Splitter.
- D. Unified Antenna exit.
- E. FLIR Blackfly S Color 5.0 MP Camera 1.
- F. FLIR Blackfly S Color 5.0 MP Camera 2.
- G. Lens Fujinon HF6XA
- H. USB 3.0 Hub
- I. Laptop connector.
- J. Power Bank 25000mAh.
- K. TW7872 L1/L2/L5 Antenna

Figure 8: All the components of the system arranged in the carrier

The implementation of the final MOBILO Carrier is made by an appropriate box is illustrated in Figure 8, which can be mounted on top of a vehicle with the appropriate magnet mounts as illustrated in Figure 9.



Figure 9: MOBILO carrier mounted on top of a vehicle

2.3 Limitations

We have extensively tested the system's performance in various conditions. Regarding movement speed, we have tested the carrier in speeds up to 60 km/h which is valid speed for our usage scenario. Furthermore, high temperature conditions seem to not affect the performance of the system, mainly due to the physical air cooling that is achieved via the

continuous vehicle movement. Finally, our design is not waterproof, and system's usage is not recommended under even low rainy conditions.

2.4 Low-cost System

Regarding the low-cost system, this can be realized by any action cams recording video data in combination with a standard geodetic GPS / GNSS such as Geomax Zenith 20 mounted in top of a carrier (Figure 10).



Figure 10: Low-cost system on top of a vehicle

Mobilo Calibration software component [D17] system should be used to calibrate the respective cameras and stereo-rig, using the standard procedure, i.e., interior and relative orientation. Linear offsets are calculated by direct measurements on the respective carrier. Data collection cannot be supported by a system such as MobiloGrabber (which only supports the advanced system) since the parts that compose it are not known upfront. Actually, this is a major drawback for the low-cost system since user is enabled to monitor the status of the GPS / GNSS and the cameras separately. Finally, data collected by the GPS / GNSS device and the respective cameras should be imported into Mobilo Data Processing tool and be manually synchronized, for each individual video file that is produced by the cameras.

3 Source Code Management

During the lifetime of a software product such as MOBILO's software components, it is common for different versions of the same source code to coexist and being written in parallel. Tracking these changes is usually an arduous and complicated task, prone to errors and cause of delays as code is constantly consolidated, tested and refactored usually numerous times during the development cycle.

This is especially prominent when the development happens across multiple cooperating teams or individual developers, all working simultaneously on different or near-identical version of the same software. Thus, the existence of a structured and systematic way to monitor the modified code is imperative for the smooth and seamless production.

The above systems in software engineering are collectively known as *version control* and they can be as simple as file name conventions corresponding to each version or can expand to dedicated version control software.

In this project we extensively used Version Control Software [2] in a distributed way based on the Bitbucket service [3]. While VCS has been employed in early development phases of our software, it was in this WP that the respective need arose imperatively: we had to work on different versions, enhance software performance and advance user interface on several aspects in parallel. Therefore, in the next sections we provide information about this kind of software, as well as info about its integration in the MOBILO Software development process.

3.1 Version Control Software (VCS)

A version control software monitors changes to all files entered under its supervision including addition of new files, deletion of existing ones and replacements. VSC allows for the cooperation of multiple developing teams on the same project which can integrate their code easier and more efficiently.

The main advantages of a VCS are: the ability to revert to previous versions of a file restoring accidental errors or deletions, testing new features of a program by branching out from the main code and provides accountability as all changes are identified by date and the person responsible for them. Most VSC can be integrated with other software development tools such as automation tools and integrated development environments.

Popular Version Control Software are Git [1] and Mercurial [4]. There are also several GUIs interfaces for VCS that visualize and manage repositories. For example, Sourcetree [5] is a GUI that simplifies the interaction with your Git repositories so the developer can focus on coding. Visual Studio also implements GUI for Git. In this project we employed Git for version control, with Sourcetree and Visual Studio 's GUIs .

3.1.1 VCS basic functions

At minimum, a VCS is expected to offer:

- **Concurrent Development:** multiple developers and design teams can work on the same set of files and source code without worrying about duplication or overwriting other team member's work. Fixes, patches and amendments can be easily developed and applied.
- **Backup and Restore:** code base files are saved as they are edited, and authors can revert to previous versions.
- **Synchronization:** Different authors can synchronize their files and stay up to date.
- **Undo:** authors current mistakes in a file by discarding changes and go back to the initial file. This can happen on a short-term time span (e.g., a working day development) or in a long-term time span (e.g., files modified over a year).

- **Track changes:** Chunks of code changes can be marked and labeled by date, author name and short messages describing the changes. By this a historical record of a project’s progress can be maintained and reviewed.
- **Branching:** Project can be cloned multiple times and developed in an isolated sandbox environment without affecting the original one.
- **Merging:** Multiple clones of the same project can be combined together, and their changes mix together.

3.2 Basic version control procedures

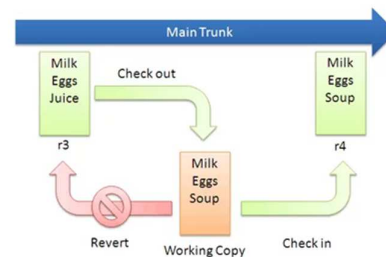
The VSC is responsible for tracking changes to anything inside a predefined storage location folder known as **repository** or repo which acts as a kind of database for the code’s files. A repository consists of the source code to be monitored and dedicated VCS files containing tables and metadata vital for the versioning operations.

A **working copy** is a developer’s personal copy of the repositories files where his work takes place. When the developer is satisfied with the changes, then it **commits** them to the repository.

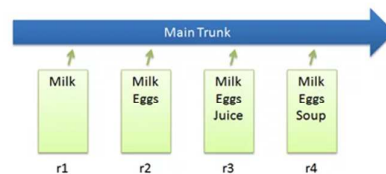
The following are a few of the most basic procedures and functions performed during version control and are automate by the BCS.

- **Add:** inserts a file in the repository for the first time. From that moment the VSC starts tracking the file’s changes.
- **Revision:** the current version a file is on.
- **Head:** the latest version of a file.
- **Check out:** VSC downloads the latest revision of the file. Check out is usually accompanying by an Edit procedure if wishing the file to be editable.
- **Check in/Commit:** uploads a file in the repository and if VCS checks any changes it applies a new Revision.
- **Update/Pull:** in the case of a remote repository, updating synchronizes the remote files with those found locally.
- **Revert:** discards local changes and reloads the latest version found in repository.

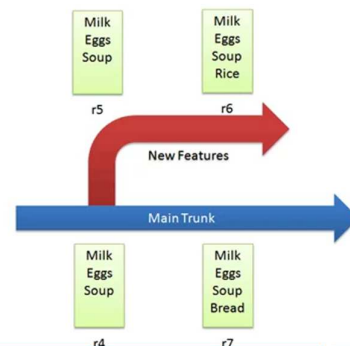
Checkout and Edit



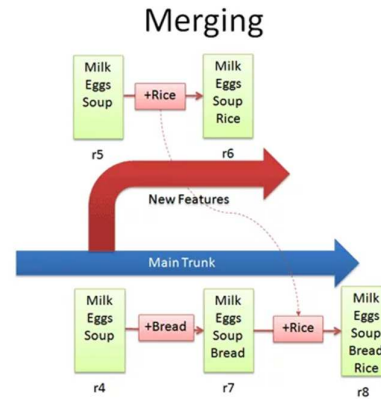
Basic Checkins



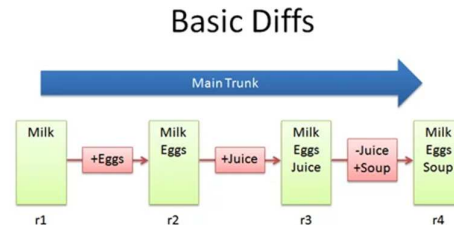
Branching



- **Branch:** creates a copy of all files and folders in the repository and starts tracking changes independently from other branches.
- **Merge:** Incorporate changes from independently development branches into a single branch.



- **Delta/Diffs:** finds the differences between two different revisions of the same file.



- **Patch:** apply changes from one file to another. This can be executed between branches of files.
- **Conflict:** records pending changes to a file from multiple versions that contradict each other and as a result neither can apply.
- **Resolve:** fixes the changes found by the Conflict operation and Checks In the correct version.
- **Lock:** prevents a file from being modified by anyone apart from the person who locked it. Used to avoid conflicts.

3.2.1 Example based on a typical scenario.

Developer X **Adds** a new file (project.txt) to the remote **repository** /Project/source/code. Afterwards she **Checks it Out**, makes modifies it by adding a few lines of code and then **Checks it in** the repository. The following day, developer Y **Updates** his local copy of project.txt and sees it with the lines of code added by developer X. He can browse the **Delta** of the file and finds out that developer X added these lines the previous day.

3.3 Types of Version Control

Version control software can be categorized into two main groups based on how they handle and distribute the central code repository.

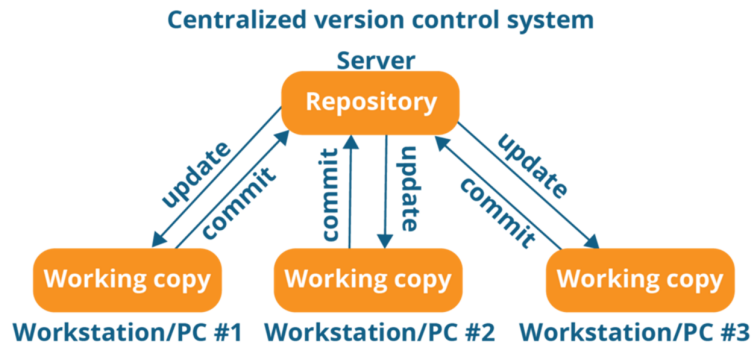
3.3.1 Centralized Version Control System (CVCS)

This type of systems maintains a central repository from which each project contributor can update his working copy of code and update it when there are changes to record. Other

contributors can see the changes and the VCS will automatically update the contents of files that were changed.

So, in a Centralized system

1. the developer commits his/hers changes to the central repository,
2. team members update their working copies with the changes recorded by the CVCS.



Benefits:

- Centralized systems are typically easier to understand and use.
- You can grant access level control on directory level.
- performs better with binary files.

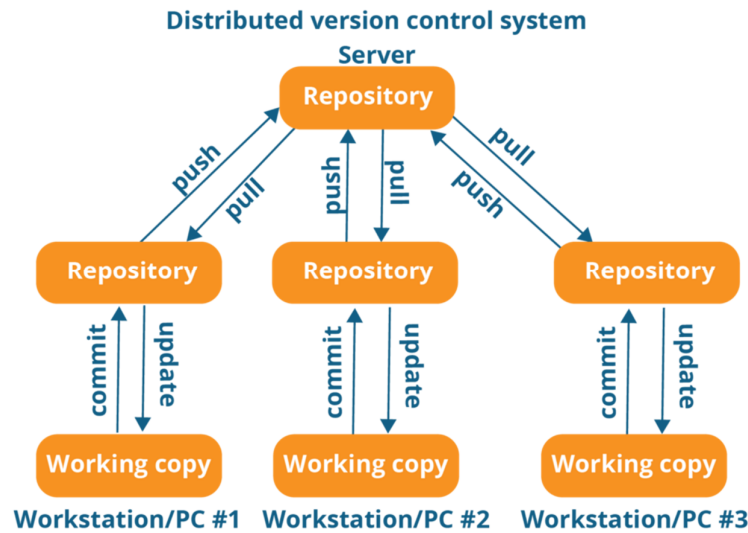
A few known CVCS are Subversion and Perforce.

3.3.2 Distributed Version Control Systems (DVCS)

In a distributed version control, every developer clones a copy of a repository and has the full history of the project on the local machine. This copy has all of the metadata of the original. The act of getting new changes from a repository is usually called **Pulling** and the act of transferring changes to a repository is called **Pushing**. A distributed version control system can also operate using a central repository which is an authoritative one.

So, in a Distributed system:

1. The developer commits his changes to his/hers local repository. Notice that at this moment no other team member has knowledge of the changes.
2. The developer pushes the changes to the remote repository,
3. Team members pull changes from the central repository on their local ones.
4. Team members update their working copies by updating from their local repositories.



Benefits:

- Performance of distributed systems is better.
- Branching and merging is much easier.
- With a distributed system, you don't need to be connected to the network all the time as a complete code repository is stored locally.

A few popular DVCS are Git and Mercurial; Among them Git is the most popular that was employed in MOBILO software development.

3.4 Git Branching Models

Since the creation of Git, users have conceived and used numerous workflows models and patterns in their Git version control aiming to achieve the best possible results in the seamless integration between software evolution, testing and building of their applications while keeping the complexity and the recording of these changes to a minimum. Some of these workflows focus mainly on handling the repository and its structure while a few others encapsulate all the workflow within the branches and the way they merge to achieve their goals. The latter are known as Git Branch Workflow models.

Some of the most popular workflows are:

- **Basic workflow:** one central repository which all contributors copy locally and then they commit back all their changes for others to use.
- **Feature Branch workflow:** for its functionality added to the project a new branch is created and all changes are included within it. When feature is completed and tested the branch merges back to the master branch.
- **Merge Requests within the Feature Branch workflow:** An expansion of the previous one which considers the differences withing the members of the development team, regarding experience, level of contribution and access rights. All changes are first

reviewed from a senior member and after his approval the changes are applied to the branches through merged requests.

- **Forking workflow:** In a forking workflow at any time a developer wants to make changes to the project, they do not copy the project directly but instead they fork it, make changes and then ask for a pull request from the owner of the repository to accept and merge. It is mainly applied to open-source projects where while there is a single owner of the central repository, the ability to change and modify the project is freely given to everyone.
- **GitFlow workflow [2]:** mainly applies to large and complex projects which require more control over their development, testing, building and release cycle. The development takes place within two main branches, the master branch which is always able for release and the development branch on which all features merge.

GitFlow workflow will be further analyzed as it was the chosen branching model applied to the development cycle of the MOBILO project.

3.5 GitFlow Workflow Description

GitFlow workflow branching model concepts and architecture was conceived in [2] and is best suited for project with multiple scheduled releases and provides a strong framework of roles and responsibilities to each development branch.

3.5.1 GitFlow Branches

The branches used in the GitFlow production environment are illustrated in Figure 11 and explained as follows:

- **Master Branch:** Master branch holds the official release history of the project and it should always be in a release state with the most current build version of the project. It is common practice all commits merge for Master to be tagged by the corresponding version number.
- **Develop Branch:** The Develop branch holds the full development and unabridged history of the project and is the origin of all Feature branches.
- **Feature Branches:** Each new feature added to the project resides within its own branch. Feature branches branch from Develop branch and they merge back to it after they have been successfully tested and build. Multiple Feature Branches can exist at the same time, one for each new feature of the project.
- **Release Branch:** When enough features have been accumulated and a new release is about to be published, a fork is made from the develop branch known as the Release branch. After its creation, only minor changes, bug fixes and documentation amendments can be committed in it. When the new release version is published, the Release branch is merged into both the Master and Develop branches. Release branches makes it possible having development teams working on publishing a current release while other teams developing new features. Also, releases are better defined in the structure of the version control.
- **Hotfixes Branches:** Hotfix branches are branches that are based on master and are the only ones allowed to fork from it. They provide a middle way between Develop and Release branches and their purpose is for applying bug fixes and patches

without interrupting the general workflow or losing time waiting for the next release. They can be considered maintenance branches that work only with Master branch.

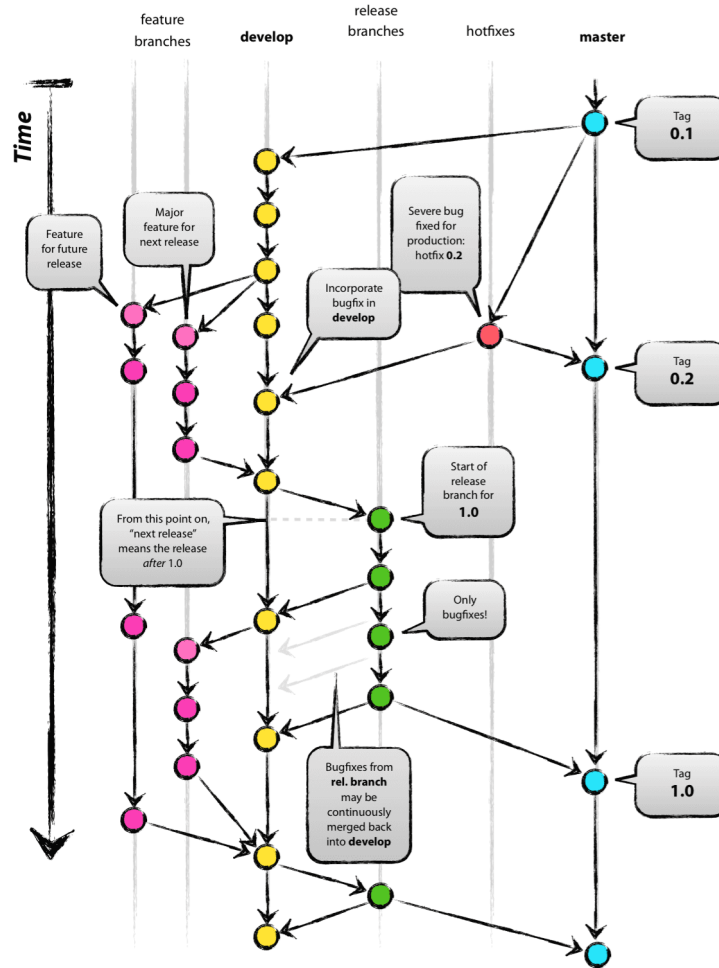


Figure 11: GitFlow example diagram

4 Integration Results – system’s accuracy

The results of the integration of our system are measured in terms of the accuracy of the provided solutions. This can be determined either theoretically by the errors provided by the several system’s parts and how they propagate in the final result, or experimentally, on a test field, with premeasured targets. The results of our experiments establish that the accuracy of the system. Specifically, there are the following aspects that need to be evaluated:

- the accuracy of the measured points,
- the precision of the measured points

The accuracy and the precision of the measured points are mainly depended on the performance of the positioning subsystem as well as the boresight misalignment calculation. As such, the float of fixed solution of the GNSS system is affecting the results, hence the precision and accuracy in urban canyons is reduced. In the following sections we first discuss about the theoretical limits of the system's accuracy and then proceed with the experimental setup and the evaluation of the system itself in terms of the accuracy achieved.

4.1 Discussion

To address the theoretic accuracy that can be achieved by our system we need to focus on the errors involved from each system's component and how this is propagated to the final result. Regarding the imaging subsystem, the two cameras are rigidly attached to a thick plexiglass, forming a stable stereo rig. There are two critical issues on this setup: the distance among the cameras and the synchronization of them. The distance among cameras (base) needs to be big enough to ensure accuracy along the camera axis, which is the axis with most uncertainty in photogrammetry. It also needs to be small enough to ensure big overlap among frames, since the stereopair is the measuring area. Objects outside the stereopair, need to be matched along different frames, which is not an optimal solution, given the uncertainty (inaccuracies) from the direct geo positioning sub system. As such the base of the system was set up 0.7m, as a good compromise among both criteria.

Given the pixel size being the measuring accuracy $\sigma_x = \sigma_y = 3.45 \mu\text{m}$, $c = 0.06\text{m}$ (Table 1), $B = 0.7\text{m}$, by using Equations 1, 2, 3, we may derive the optimistic Table 2. Nevertheless, Table 2 provides a measure of the expected accuracy, in relation to the camera-object distance.

$$\sigma_X = \frac{H}{C} \sigma_x \quad \text{Eq.(1)}$$

$$\sigma_Y = \frac{\sqrt{2} H}{2 c} \sigma_y \quad \text{Eq.(2)}$$

$$\sigma_Z = \frac{H^2}{cB} \sigma_{px} \quad \text{Eq.(3)}$$

Regarding the imaging subsystem, the theoretic accuracy given the system's components is given [6] as a combination of pixel size, focal length, distance between cameras the accuracy of measurement inside each image. In our system we have the following values

- Pixel size 3.45 μm
- Focal length 6 mm
- Measurement accuracy: 1 pixel (theoretical)
- Distance between cameras in the stereo pair 0.68 m

The results are summarized in Table 2, where becomes clear that the accuracy of the imaging subsystem in distances greater than 20 m becomes less than 23 cm in the geodetic reference system. Moreover, in the case where measurement accuracy becomes 2 pixels, the respective values of Table 2 are doubled, for example when matching algorithm [7] fails

to automatically recognize the point of interest. Following the realization that objects beyond 25m would already have significant photogrammetric error, just from camera capturing system, the stereo frames must be captured at distances of 20m or less, to ensure that all potential objects of interest should be at a distance <25m from cameras, in at least one stereo frame.

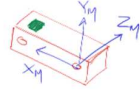

	Camera XYZ			Geodetic XYZ	
					
Distance [m]	σ_{Xc} [m]	σ_{Yc} [m]	σ_{Zc} [m]	σ_{XY} [m]	σ_Z [m]
5	0.003	0.002	0.015	0.015	0.002
10	0.006	0.004	0.060	0.060	0.004
15	0.009	0.006	0.135	0.135	0.006
20	0.012	0.008	0.239	0.239	0.008
25	0.014	0.010	0.374	0.374	0.010

Table 2: Accuracy of the stereo rig.

On the other hand, the accuracy of the positioning subsystem is dominated by the accuracy of the GPS / GNSS provided positions and the rotations provided by the INS / IMU. According to the boards specifications and our experiments these are

- RMS Roll/pitch 0.5°
- RMS Yaw 1.5°
- RMS X/Y (fix) 3cm
- RMS Z (fix) 5cm

Applying these on a typical 3d transformation according to the Figure 12 we obtain the values displayed in Table 3 regarding the estimated error in the final objects position, only by the positioning subsystem.

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = R_{\Omega\Phi\kappa} * \begin{bmatrix} X_M \\ Y_M \\ Z_M \end{bmatrix} + \begin{bmatrix} X_0 \\ Y_0 \\ Z_0 \end{bmatrix}$$


Figure 12: Geodetic vs photogrammetric reference systems

Distance [m]	σ_{Xins} [m]	σ_{Yins} [m]	σ_{Zins} [m]	σ_{XYins} [m]	σ_Z [m]	σ_{XY} [m]
5	0.044	0.131	0.044	0.138	0.066	0.141
10	0.087	0.262	0.087	0.276	0.101	0.278
15	0.131	0.393	0.131	0.414	0.140	0.415
20	0.175	0.524	0.175	0.552	0.182	0.553
25	0.218	0.654	0.218	0.690	0.224	0.691

Table 3: Positioning subsystem error transmission

Obviously, the RMS of 1.5° in the estimation of Yaw heavily affects the horizontal accuracy of the provided solutions, adding more than half a meter on the error of the objects found at lead 20 meters away from the MOBILO system. This is the reason for establishing a more accurate solution presented in WP4 which allows the calculation of heading from consecutive fixed positions. Substituting thus in the calculation of Yaw an estimated RMS error of 0.35° (instead of the 1.5° provided), we obtain the values of Table 4 regarding the estimated error in the final objects position, only by the positioning subsystem.

Distance [m]	σ_{Xins} [m]	σ_{Yins} [m]	σ_{Zins} [m]	σ_{XYins} [m]	σ_Z [m]	σ_{XY} [m]
5	0.044	0.031	0.044	0.053	0.066	0.061
10	0.087	0.061	0.087	0.107	0.101	0.111
15	0.131	0.092	0.131	0.160	0.140	0.163
20	0.175	0.122	0.175	0.213	0.182	0.215
25	0.218	0.153	0.218	0.266	0.224	0.268

Table 4: Positioning subsystem error transmission with heading calibration

As such, the estimated accuracy is obtained by combining the errors provided in Table 2 and Table 4, resulting in an estimated value of $\sigma_{XY} = \pm 0.46m$ and $\sigma_Z = \pm 0.23m$ in objects found in a distance of 25 m meters from the cameras, given that GPS provided positions are fixed.

Finally, regarding the synchronization error of the advanced system, to ensure that cameras were properly synchronized a simplistic test took place. The two cameras were setup looking an on screen timer while capturing synchronized frames (Figure 3). As they both record the same time with up to 1/100 sec division, it is safe to assume that the synchronization is at least that good. Therefore, at speed of 50km/h, this time is interpreted in 0.14m distance and at speed of 80km/h, 0.22m. Such differentiations from the calibrated relative orientation of the cameras would have provided suboptimal final results. According to system verification, where RMS on the absolute position of the points is much better, the synchronization of the cameras must be much better, at the rank of 1/1000 of the sec or better, where the relative calibration would be affected by 0.014m at 50km/h, mean speed during capture of the evaluation data.

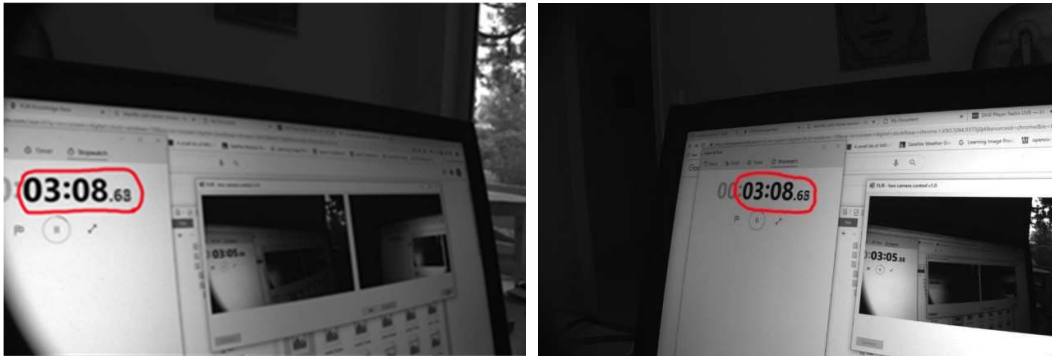


Figure 13: Synchronization test among the cameras, with the GPIO cable

Regarding the low-cost system, where video synchronization is performed manual, and system's calibration is not known in advance, it is expected to incur higher errors due to the temporal analysis of the video. For example, given a vehicle's speed of 60 km/h, and a video of 30 FPS, the distance between consecutive video frames would be approximately 55 cm. As such even with a very good synchronization down to the half of the frame rate, would introduce an additional error of 27 cm. Moreover, given that usually action cams incur high value of radial distortion, the values of Table 2 should be significantly promoted, downgrading the overall system's accuracy.

4.2 Experimental Setup

To test the validity of the system, one georeferenced check field with physical targets has been measured using RTK GPS/GNSS inside a suburban area, and a total of 41 check points (Figure 14).

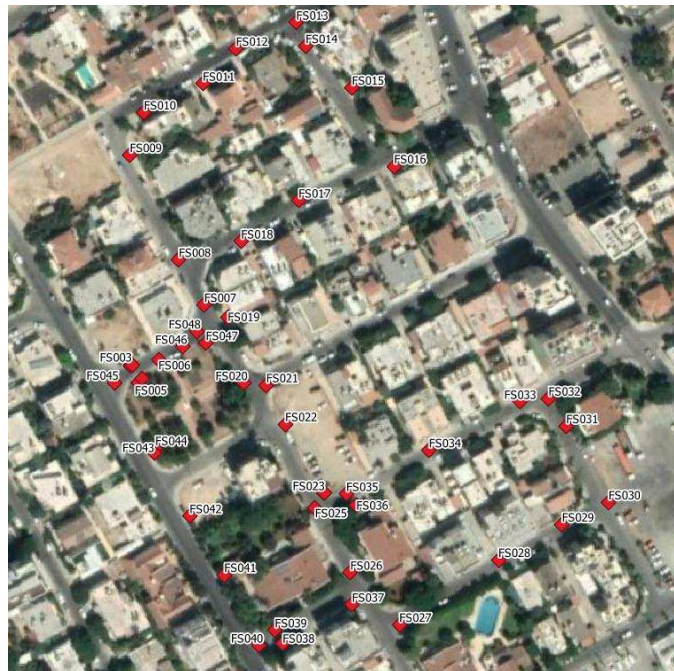


Figure 14: Control points in the test field

Camera interiors and stereo rig relative orientation were determined using the developed software (Mobilo Calibration) and the techniques of [8]. The results of the calibration process are summarized in the following Table 5. The respective calibration results demonstrate that accuracy is typical of the corresponding photogrammetric processes (i.e., 0.5 pixels for interior and 1 pixel for relative). Therefore, we can assume that both equipment and methodologies followed are proper and results within expected limits.

	Left Camera Interior	Right Camera Interior	Relative*
number of Images	95	103	47
number of observations	8360	9060	4136
Residuals (pixel)	0.504	0.524	0.94

*pair of images

Table 5: Interior and relative orientation results.

Having completed the system calibration, MOBILO system was initialized on top of the car and then performed at least two passes in each one of the control fields following the same and opposite directions (Figure 15).



Figure 15: Test field with MOBILO trajectories

4.3 Results

The collected data were processed by a user and the predefined points were recognized and digitized on the software, to compare their computed coordinates with the ones of the

check field. All digitized points were at an effective distance from the cameras of no more than 20 m; this is a reasonable assumption since consecutive pairs of frames are not expected to be separated more than 22 m, even at working speed of 80 kph and a frame rate of 1 sec. The results of the comparison between check points and calculated coordinates are summarized in the following Table 6.

	# points	# observations	XY (m)	Z (m)
St Dev	41	78	0.50	0.45
St Dev (only fixed)	40	40	0.38	0.43

Table 6: Accuracy of solutions

Clearly the average positional error is less than 0.50 m horizontally and vertically in all conditions, that is, in all 41 measured points observed by 78 individual positions (video frames). Furthermore, if we isolate the frames where we have fixed solutions, the error in XY reduces to 0.38 and in Z to 0.43. Moreover, the horizontal error is mainly concentrated along the vehicle movement axis (parallel to camera axis), with the maximum error along the stereo rig's X axis not exceeding 0.25 m at the worst case. This is an expected result given the fact that photogrammetry suffers of errors in depth calculation, and the relatively small baseline does not support better results.

	# points	# observations	XY (m)	Z (m)
Mean of St Dev	30	67	0.41	0.44
Mean of St Dev (only fixed)	10	23	0.15	0.25

Table 7: Repeatability of solutions

The second evaluation concerns the repeatability of the solutions. According to Table 7 the repeatability was evaluated over 30 total points with 67 observations. Initially we calculated the standard deviation of all observations for each individual point. Then, values displayed in Table 7 are the average of all standard deviations for all points.

From all above presented results becomes clear that the system's accuracy conforms to the theoretical expectations regarding the error in the calculation of the on-image recognized and digitized objects' positions.

5 Licensing

All developed software components in the project are protected and locked with a licensing system that is based on a service running the cloud. The licensing mechanism locks the software over specific hardware properties of a workstation, providing at the same time the ability to transfer the license between workstations. The license check is performed over the web and on a database that stores client and software data. While this mechanism was already present at project's startup, we proceeded to its complete refactoring in order to improve several aspects of its security and also provide an easy way to manage licenses. We have therefore developed the LiveLocker system which is described in following paragraphs.

LiveLocker system constitutes a complete DDRM (Distributed Digital Rights Management) system for software developers that wishes to have remote access control to their propriety applications. LiveLocker system main functions are:

- Database functionality for recording applications versioning and customers' information.
- Graphical UI for managing system's database.
- Issuing and distributing licenses to application users.
- License validation and authorization per machine installation.
- Trial application mode.
- Application offline license mode.

The architecture of LiveLocker System is illustrated in Figure 16, and is composed of the following components:

- Amazon hosted virtual machine running Windows Server OS.
- IIS Web Server for hosting the system's server-side functionality.
- Sql Server for database.
- Customer Manager GUI, a windows form application for managing the database.
- LiveLocker windows library which is the front end of the LiveLocker License System.
- The propriety application that is being monitored by the LiveLocker library.

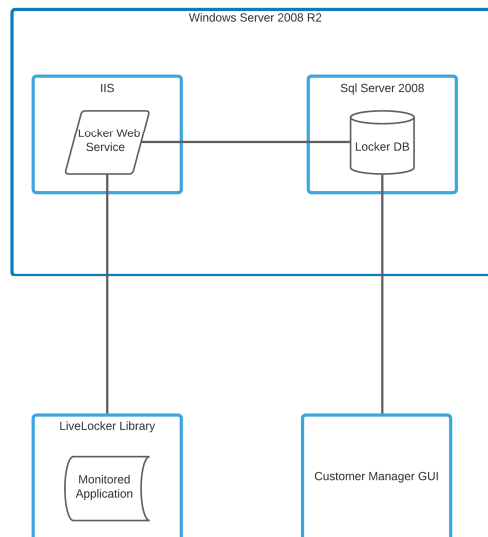


Figure 16: LiveLocker system architecture

5.1 General Description

Applications that need to be locked consume LiveLocker client library which performs the following actions in order to establish authentication and authorization for the supervised application:

1. LiveLocker client instance collects all the necessary customer information and unique hardware characteristics that are necessary for application and license identification.
2. Data are sent to the LockerService web service.
3. Web service connects to database and validates credentials and machine.
4. Web service responds to LiveLocker client the supervised application's license status.
5. If license is valid, LiveLocker client authorized its use, otherwise a trial mode may be offered to client.
6. In case of internet or server connection failure, and only if a valid license has already been activated in the current machine, LiveLocker client offers an offline mode for a limited number of days.

In the backend an administrator user is responsible for the following tasks, through the Customer Manager GUI:

1. Creates and inserts into database a new license.
2. Links new license to customer through customer's vat.
3. Links new license to supervised application through application's database id.
4. Assigns to license a registration date and a possible expiration date.
5. Sends the license serial number to customer.

LiveLocker API software has been built using the .NET platform and is written in C# programming language (in various version). Its development has been organized in a single solution container (LiveLocker solution) which hosts three software related projects. Visual Studio 2019 was the preferred development IDE. LiveLocker solution constitutes of the following projects:

- **LockerService project:** a WCF SOAP service that handles LiveLockerClient request and communication with the database (Section 5.2).
- **LiveLockerClient project:** a .net framework library that provides license validation and authorization functionality withing the monitored application (Section 5.3).
- **Customer Manager project:** a back end .net framework windows form application that manages the database of the service (Section 5.4).

Apart from the in-house libraries the following libraries where used:

- **Entity Framework Core v6** as the default object-database mapper.
- **MailKit** for establishing communication with the smtp server.

5.2 LockerService Project

The LockerService includes all the back-end services of the LiveLocker API. It is a Windows Communication Foundation (WCF) project hosted in IIS server and utilizes SOAP over HTTPS transport protocol as well as message level encryption. LockerService uses ASP.Net framework 4 as its development platform. The following classes – data contracts are used:

- **ClientInfo DataContract:** ClientInfo class type acts as container for customer information and smtp configuration data transfer from LiveLockerClient front end library to the LiveLocker service back end.

Property	Type	Description
AppNameId	Int	The monitored application name id
AppVerId	Int	The monitored application version id
CpuId	String	The cpu id of the hardware which executes the monitored application
MacAddress	String	The macaddress of the hardware which executes the monitored application
VatId	String	License owner VAT
Sn	String	License serial number
Email	String	License owner email
RequestTrialEmail	String	Email provided for trial confirmation code
TrialConfirmationCode	String	Trial confirmation code send to RequestTrialEmail
TrialPeriod	Int	Trial period in days
DeactivationDaysThreshold	Int	Minimum time period after which remote license parking is allowed. Period in days.
SmtpServer	String	The smtp server for sending the RequestTrialEmail.
SmtpUser	String	The smtp server user.
SmtpPwd	String	The smtp server password.
SmtpPort	String	The smtp server port.
SmtpSenderName	String	RequestTrialEmail sender address.

- **TrialResponse DataContract:** TrialResponse class type returns the data produced by the LiveLockerClient trial request.

Property	Type	Description
TrialStatus	Enum	The trial status of the license: <ul style="list-style-type: none"> • NoTrial: license has no ongoing trial • TrialOngoing: a trial license is active • TrialUnconfirmed: a trial license exists but is yet unconfirmed • TrialExpired: trial license period has expired • LicenseExists: a valid regular license exists
DaysLeft	Int	Days left for trial license to expire
TrialRequestEmail	String	The email to which the confirmation email has been sent

All service operations are declared in the IService interface and implemented in Service class. All operations use a single ClientInfo type object for parameter. Operations are summarized in the following table.

Operation	Return type	Description
GetLicenseStatus	LockStatus	Determines the license status of the current customer-machine combination.
RegisterPermanent	LockStatus	Attempts to register a valid license.
ParkCurrentMachine	LockStatus	Deactivates license from the current machine.
ParkNonCurrentMachine	LockStatus	Deactivates license from non-current machine.
GetTrialLicenseStatus	TrialResponse	Gets the status of a trial license if it exists of the current customer – machine combination.
SendConfirmationEmail	TrialResponse	Sends the confirmation code after a trial license request.
ConfirmTrialCode	TrialResponse	Confirms trial license code send application.
Test	String	Test operation for ensuring LiveLockerClient can communicate with LiveLockerService.

5.3 LiveLockerClient Project

LiveLockerClient library is hosted in the licensed application and authorizes its use by the customer. It connects to the database through the LiveLockerService endpoints and validates the customers license credentials. Setup of locker is done through the monitored application code. LiveLockerClient uses the .Net Framework 4 as its development platform. The application that consumes the LiveLocker library, e.g., *Mobilo Data Processing* application, is responsible for importing the following information:

- Customer’s vat number,
- Customer’s license serial number,
- Supervised Application database name Id and version id.
- SMTP email server credentials
- Trial period in days if customer wishes to activate the supervised application in trial mode.
- Time period in days prior before which an activated machine is eligible for remote deactivation

The following code in Figure 17 demonstrates how a LiveLocker instance can be constructed and used for controlling access to the supervised application.

LiveLocker supports also offline licensing based on the following algorithm presented in Figure 18. The algorithm initially checks whether an internet connection exists and if so it follows the regular licensing procedure where the server is asked for license status and saves some *offline data* to the disk so as to be possibly used in the future. These data are used in the case an internet connection is not present; in this case LiveLocker checks whether offline data are present, and whether the time period that has passed offline is valid.

```

using LiveLockerClient.AppCode;
using LiveLockerClient.AppCode.Components;

static void Main(string[] args){
    ApplicationInfo app = new ApplicationInfo(appId: 4, appVer: 2);
    Vat vat = new Vat("123456789");
    SerialNumber sn = new SerialNumber("TST00001");
    Assembly assembly = Assembly.GetExecutingAssembly();
    SmtpConfiguration smtp = new SmtpConfiguration(
        "smtp.server.com", "smpty_email", "smtp_pwd", port:111, "sender", "sender@email.com");
    TrialConfiguration conf = new TrialConfiguration(smtp, trialPeriod: 15, deactivationDays: 3);

    Locker locker = new Locker(app, vat, sn, conf, assembly, "eula.pdf");
    bool unlock = locker.UnlockApplication();

    SupervisedApplication app = new SupervisedApplication();
    if (unlock)
        app.Execute();
}
    
```

Figure 17: Sample Code

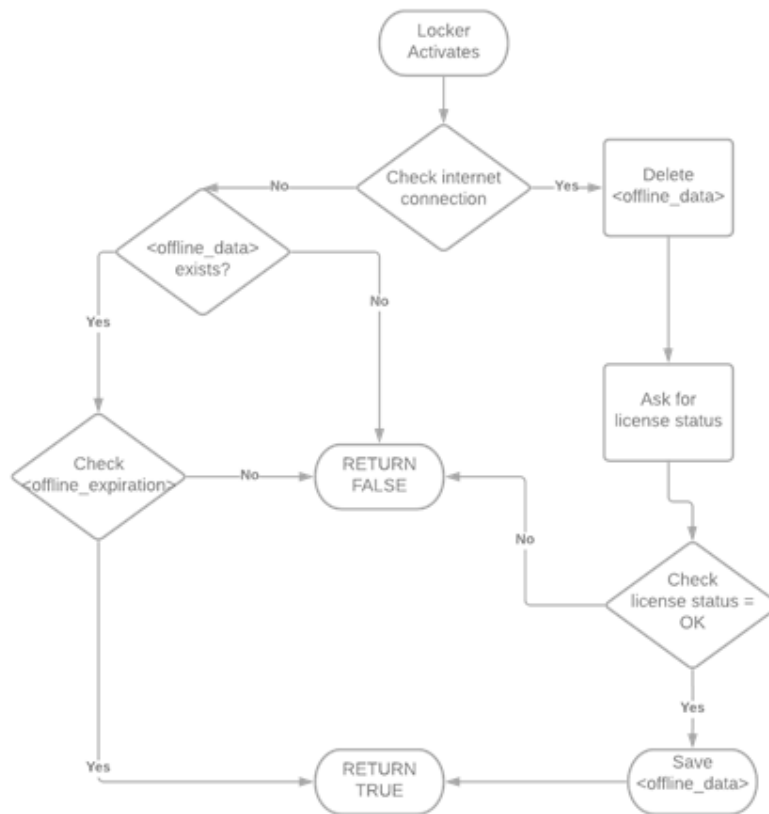


Figure 18: Locking algorithm with offline check

5.3.1 User Interface

LiveLocker includes a single class responsible for providing the graphical interface through which user can provide his/hers license credential for validation and authorization.

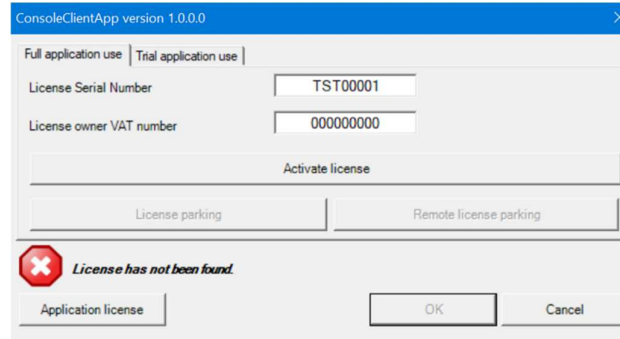


Figure 19: FrmLocker UI

5.3.2 API Usage

The LiveLockerClient dynamic link library contains a single public class Locker which exposes the LiveLocker API. Locker is responsible enabling the monitored application use and initiative the procedures for trial or offline mode.

Class Member	Member Type	Type	Access	Description
LicenseStatus	Property	LockStatus	Public Get	Holds the license current status based on LockStatus enum.
TrialResponse	Property	TrialResponse	Public Get	Holds the LockerService response after a request for trial license.
Info	Property	ClientInfo	Public Get, only constructor initialization.	Holds the customer, application and current machine information.
EulaFile	Property	String	Public Get, only constructor initialization.	The user license agreement pdf file that can be viewed through the FrmLocker GUI.
Locker	Constructor		Public	Public constructor
UnlockApplication()	Method	Bool	Public	Asks customer credentials and authorizes the usage of the monitored application.
ShowLicenseInformation()	Method	Bool	Public	Activates the FrmLocker GUI for viewing and manipulating the license information.

Livelocker is consumed by our MOBILo projects so as to provide valid licenses.

5.4 Customer Manager Project

Customer Manager is a window forms application which purpose is the back-end manipulation of the LiveLocker system database. Through its GUI user can insert, edit, or delete customers, licenses, trial licenses and application details such as version and id. User of Customer Manager application are only those who have valid credential on the LiveLocker System Database. Model classes are autogenerated by the EntityFramework ORM through the Visual Studio IDE interface.

5.4.1 UI classes

GUI is composed by the following three classes – forms.

Class FrmMain (Figure 20): the main interface of the CustomerManager application. It is composed from two data grid views where the contents of licenses and machines are displayed. From here, user can perform basic insertion and editing action. Customer’s information is shown, and filtering is done by customer attribute.

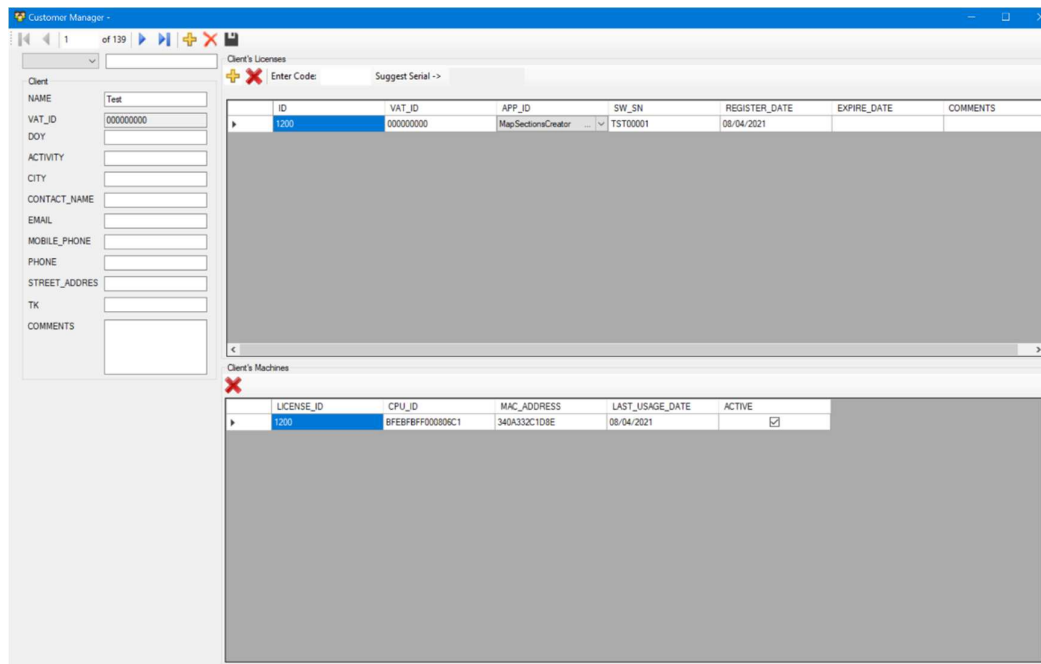


Figure 20: Customer Manager Main Form

Class ClientInfo (Figure 21): class creates a small dialog form when user needs to insert a new customer in the database. Validates the uniqueness of the VAT and customer name.

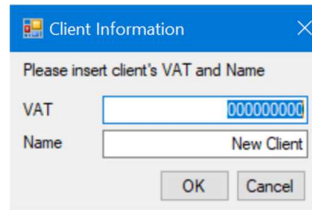


Figure 21: Customer info dialog

Class FrmServerDialog (Figure 22): class presents the dialog for database connection string information and for validating its use.

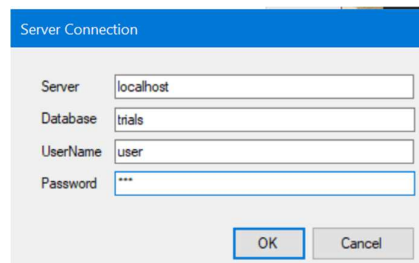


Figure 22: DB connection

6 Conclusions

MOBILO general architecture consists of a Mobile Mapping System that includes several components and is able to be mounted on top of vehicle in order to collect georeferenced video data, as well as three software components which are used to calibrate the system, collect and process the actual data. In this deliverable, we focus on the task of MOBILO System Integration. We describe the evolution of the MOBILO System Carrier, preliminary designs and issues appeared, version control software management strategies that helps in software integration and software licensing methods that are essential for commercial software exploitation.

Our final system architecture regarding the advanced system is based on a rigid Plexiglas box hosts all parts in its interior, is attached to the carrying vehicle with magnetic mounts and provides connection with a single cable to the managing computer. Moreover, the results of our experiments establish that the accuracy of the system is below 0.50 m which outperforms our initial expectations. The low-cost system can be implemented by any carrier capable of mounting a standard GPS / GNSS along with two action cams; however, the accuracy of the results will suffer from the high synchronization and stereo-rig calibration error (at least 2 times higher than the one of the advanced system).

There are also two other issues that emerged during system integration: The employment of VCS enables the continuous development of our system, while the complete refactor of an existing licensing mechanism led to a complete DDRM (Distributed Digital Rights Management) system for software developers that enable remote access control to their propriety applications.

We have to note here that the cost of all system components, including wires, rig and magnetic base mounts, is below 3000€, which is exceptionally low for an MMS. Nevertheless, the integration of all these under a turnkey solution was not straight forward: the commercial ready system cost should include labor, system and software development, maintenance and support, and further development costs.

As emerges by the integration, given that our system will provide points with 0.5m uncertainty in absolute coordinates, it emerges as a nice alternative to more expensive MMS for several applications. It is aiming infrastructure recording and monitoring, as well as economical solution for recording road condition, or even simple recording of image sequences with high quality positioning so that revisiting or even direct measurements are possible. Examples of use:

- Traffic sign recording for GIS input
- Road surface condition recording and estimation of area to be replaced.
- Façade measurements and estate condition
- Electricity, telecommunication and light poles recording, along with equipment type
- Section recording of old provincial roads, for maintenance or redesign (paved shoulders' width)
- Extensive photo recording of municipality roads, with georeferencing for easy documentation/archiving of areas/points of interest.

7 References

- [1] "Git - fast version control," [Online]. Available: <https://git-scm.com/>. [Accessed 31 03 2021].
- [2] V. Driessen, "A successful Git branching model," 05 01 2010. [Online]. Available: <https://nvie.com/posts/a-successful-git-branching-model/>. [Accessed 31 03 2020].
- [3] "Atlassian Bitbucket," Atlassian, [Online]. Available: <https://bitbucket.org/product/>. [Accessed 31 03 2021].
- [4] "Mercurial," [Online]. Available: <https://www.mercurial-scm.org/>. [Accessed 31 03 2021].
- [5] "Sourcetree," Atlassian, [Online]. Available: <https://www.sourcetreeapp.com/>. [Accessed 31 03 2021].
- [6] T. Schenk, Introduction to Photogrammetry, 2070 Neil Ave., Columbus, OH 43210: Department of Civil and Environmental Engineering and Geodetic Science, The Ohio State University, 2005.
- [7] "OpenCV software library," [Online]. Available: [1] <https://opencv.org/>. [Accessed 31 3 2019].
- [8] Z. Zhang, "A Flexible New Technique for Camera Calibration," Technical Report MSR-TR-98-71, Microsoft Research, Microsoft Corporation, One Microsoft Way, Redmond, WA 98052, 2008.