SEVENTH FRAMEWORK PROGRAMME	**Project Acronym: GiraffPlus** **Project Title: Combining social interaction and long term monitoring for promoting independent living** **Grant agreement no.: 288173** **Starting date: 1st January 2012** **Ending date: 31st December 2014**

D2.3 Third Prototype of sensors, Giraff platform and network system report

WP related to the Deliverable:	2
Nature:	P
Dissemination Level :	PU
Version:	2
Author(s):	Filippo Palumbo (CNR-ISTI), Davide La Rosa (CNR-ISTI), Francesco Potortì (CNR-ISTI), Erina Ferro (CNR-ISTI), Michele Girolami (CNR-ISTI), Ales Stimec (XLAB), Maria Lindén (MDH), Gregory Koshmak (MDH), Eleni Odontidou (Giraff), Javier Gonzalez (UMA), Cipriano Galindo (UMA)
Project Participant(s) Contributing:	CNR-ISTI, XLAB, MDH, Giraff, UMA
Contractual Date of Delivery:	20140701
Actual Date of Delivery:	20140701

Document History

Version	Date	Type of editing	Editorial
0.0	13/06/14	Table of Content and Initial Draft	CNR-ISTI
0.1	26/06/14	First round of contribution integrated	CNR-ISTI
0.2	06/07/14	Addition of CNR-ISTI and UMA contribution	CNR-ISTI
1.0	09/07/14	Complete version before internal review	CNR-ISTI
1.1	14/07/14	Revised version after internal review	CNR-ISTI
Final	15/07/14	Final version of the document	CNR-ISTI

Disclaimer:

No confidential material is included therein.

Deliverable Summary

This document reports on the third prototype of the system that is currently deployed in all test sites.

The third prototype at month 30 provides the final version of the network system and the second and final version of the Giraff platform. A middleware solution helps to integrate the software components developed by the WP3 and WP4 and enhanced Giraff platform with safer and semi-autonomous mobility features.

The prototype includes the Giraff robot, the Look4Myhealth kit, the monitoring sensors from Tunstall, additional environmental sensors, a physiological sensor for pulse oximetry measurements based on Android, the context recognition and configuration planning modules, and the remote storage and repository to collect user data.

Additionally a new wearable sensor containing an inertial system is also introduced. It can be carried out by the primary user and used to give additional information regarding the user activities.

The final version of the middleware is presented and the semi-autonomy features for the Giraff robot are described.

Table of Content

List of Tables

List of Figures

1 Introduction

1.1 Scope of the document

The document is a progress report with respect of the D2.2 and describes the third prototype of sensors, Giraff robot, and the GiraffPlus integrated system. The final version of the sensor network system is provided together with second and final version of the Giraff platform. There have been improvements on all parts of the system and novel features additions. In particular, the stable version of the middleware has been released in its desktop and mobile adaptation, allowing new features as heartbeat, failover system, and dynamic reconfiguration. New wearable inertial system has been investigated and ready to be integrated. Finally, the Giraff platform in its improvements is presented allowing collaborative control between the human teleoperator and the onboard motion.

A considerable effort has been spent during this period on providing stability and robustness of the system. In particular, the results of stress tests performed on sensors readings latency and middleware performance are presented.

This third prototype of the GiraffPlus system is currently under deployment at 15 test sites.

1.2 Deliverable structure

The document starts with an outline of what has been achieved and how it matches the DoW. In section 2 the sensor network system is described, section 3 focuses on the middleware infrastructure, while section 4 is dedicated to the Giraff platform and its collaborative control feature. Each section presents the effort made in terms of new features, tests, and bug fixing.

1.3 Deviations with respect to the plan

The project is proceeding according to the plan outlined in the DoW with no deviations. In particular, Table 1 shows how the requirements of the DoW are matched to what has been achieved by month 30 in the project.

Table 1 Matching requirements from DoW with results achieved at M30

Required from the DoW	Achieved at month 30
Sensor network design and implementation: The network infrastructure is refined according to the needs of other WPs	Section 2 The solutions adopted in order to allow plug-and-play use of sensors/actuators and other equipment already available on the market have been consolidated and tested. The network infrastructure has been refined according to the needs of the other

	WPs. The refinements to the main components of the sensor network system, the tests made in order to guarantee the robustness of the sensor network are described in section 2.2.
Middleware design and implementation: The final version of the middleware is delivered	Section 3 The final version of the middleware has been delivered al month 30 and presented in an international peer reviewed journal, featuring new functionalities and hardware and operating system compatibilities. New features regarding the heartbeat, failover and dynamic reconfiguration mechanisms are described in section 3.2. The tests performed in order to guarantee robustness and scalability of the middleware infrastructure are described in section 3.3.
Development of Giraff robot platform: A new version of the platform is delivered	Section 4 The new software version (v2.4) for both Giraff and Pilot developed and released during this period includes new features for the 4.0 version Giraffs which are related with the night vision and height adjustment. A lot of changes and bug fixing has been made and described in section 4.2 and Appendix 1.
Additional sensor selection and design: A complete version of the sensor system ready for the last evaluation	Section 2 Additional inertial wearable sensors were selected to be integrated with the Android mobile version of the GiraffPlus platform. After an accurate review of the commercial available solutions, the Pebble SmartWatch was selected as the optimal choice (Section 2.3.1).

	Android-based sensor platform has been successfully integrated into the GiraffPlus platform. As a result, MDHPulseOximeter and MDHFallDetector were added to the system (via Engineering GUI) as complementary sensors (Section 2.3.2). Additional control functionalities were introduced, including wireless network availability, average activity check and "switch on/off status" notifications.
Enabling safe and secure communication: An intermediate version is available	Section 3 A continuous maintenance activity was carried out during the period and, since the deliverable D2.2 some bug fixing were performed on the libraries used in order to guarantee safe and secure communication between services (Section 3.3.1).
Improvement of Giraff mobility: Giraff platform collaborative control between the human teleoperator and the onboard motion system to facilitate passing through doors, corridors and clutter spaces	Section 4 According to the requirements for the third prototype of the GiraffPlus system at month 30, the robotic platform has been enhanced with a collaborative control system, which consists of a high-level controller that extends the capabilities of the prototype presented in deliverable D2.2. It complements the motion commands from the driver with the semiautonomous navigation system to safely pass through doors, corridors, and cluttered spaces.

2 The Sensor Network System

The solutions adopted in order to allow plug-and-play use of sensors/actuators and other equipment already available on the market have been consolidated and tested. The network infrastructure has been refined according to the needs of the other WPs. The refinements to the main components of the sensor network system, the tests made in order to guarantee the robustness of the sensor network, and the additional sensors selected and integrated are described in the following subsections.

2.1 Main components

Environmental sensors provided by Tunstall, physiological sensors provided by Intellicare, and additional ZigBee and wearable sensors are integrated in the GiraffPlus system by means of gateways [1] exploiting the capabilities of the GiraffPlus middleware as described in D2.2.

The gateways deployed to the set-top boxes and android tablets installed in the primary user house, listens for sensed data, and publishes the information to the context bus when an event is raised.

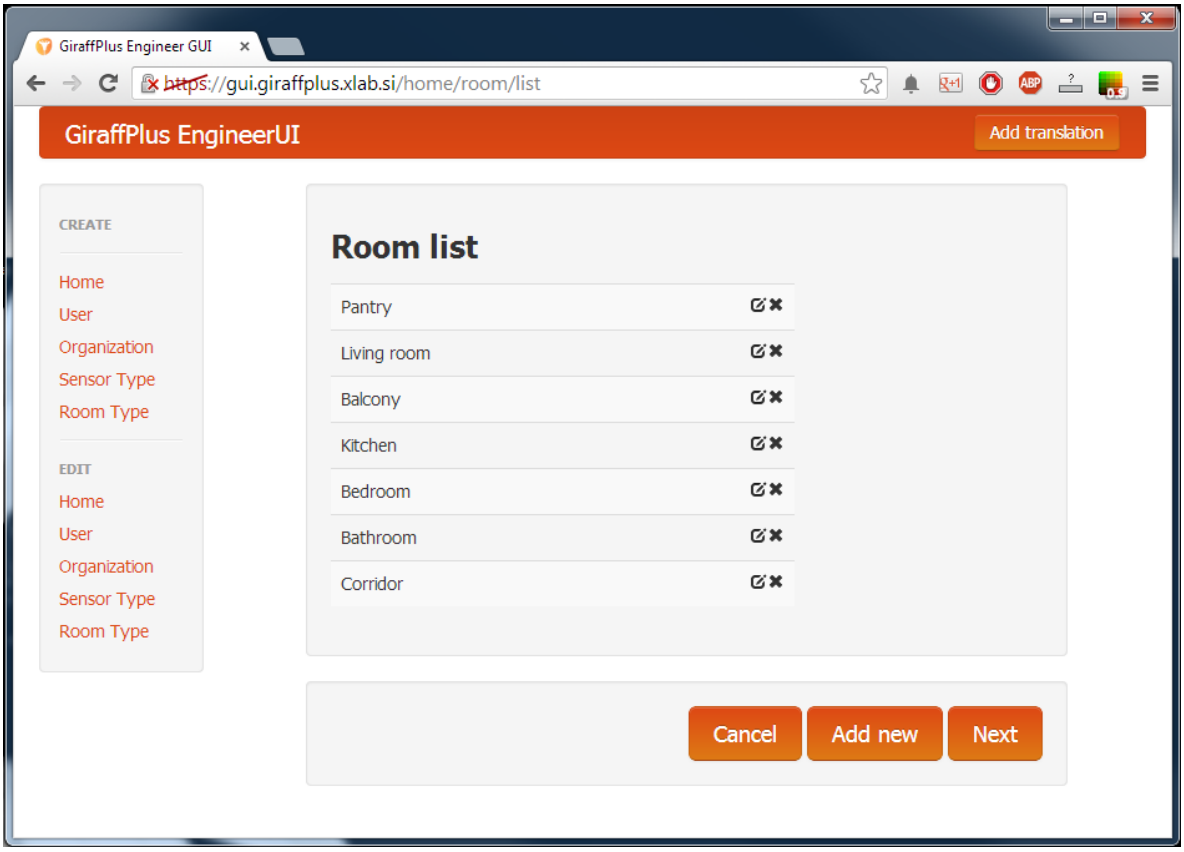


Figure 1 A screenshot of the GiraffPlus Engineer GUI

The first configuration of a newly installed sensor is made through the GiraffPlus Engineer GUI (Figure 1). It enables the GiraffPlus professionals to enter and edit configuration data

for any home included in the GiraffPlus ecosystem. The GUI is a wizard-like web application, which can be used from any web-enabled device using any web of the major web browsers. It was implemented in Java using the Play Framework [2]. In the backend it communicates securely with the GiraffPlus Storage Web Service to store all configuration data in the underlying database.

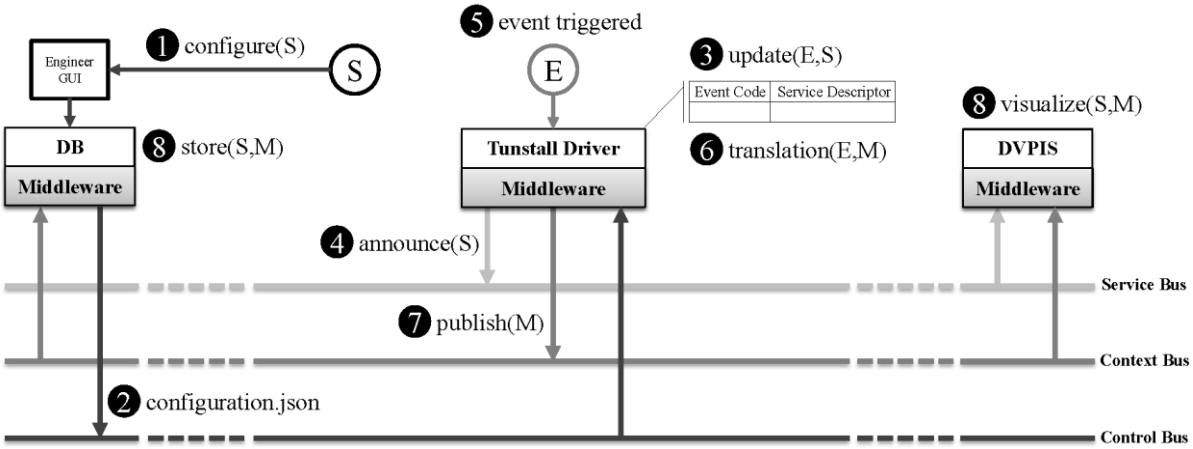


Figure 2 The sensor integration mechanism

Figure 2 shows the steps involved in the integration of Tunstall gateway and how its data format is translated into the data schema managed by the middleware. The same steps are performed for other sensors integrated in the system. When a new sensor is installed in the house, it is configured by means of the Engineer GUI (Step 1), the Server Infrastructure middleware instance publish on the Control Bus topic:

<<location>>/controlBus/configuration

a JSON file (configuration.json) containing the new configuration (Step 2). Since the Tunstall sensor data schema is composed of several event codes representing all the possible events that a sensor can trigger, the Tunstall driver component developed upon the GiraffPlus middleware maintains in memory a table that associates a possible event to a sensor, described in terms of *ServiceDescriptor*. This table is updated with the information about the new configuration forwarded by the middleware (Step 3) and the announce method is called on the Service Bus notifying all the interested service about the presence of a new sensor (Step 4). When an event is triggered by the new sensor (step 5), a translation from event code to the GiraffPlus message format is done (Step 6) and the message is published on the Context Bus topic composed of the keyword ”sensor” and the configured ”id” (Step 7). Once the message is published, each component that was subscribed on that topic will receive the message (Step 8). This is the case of DVPIS, monitoring real time data and alarms, and GiraffPlus MQTT Listener, storing historical context data for future long term data analysis. These steps are also made when a sensor is removed or changed in its configuration.

2.2 Guaranteeing the sensor network robustness

One of the main goals of the GiraffPlus system is the possibility to monitor in the long-term period the activities and behaviors of the user while he/she interacts with the environment. In this context, the reliability of the sensor network plays a crucial role. For this reason, we performed several tests to determine the performance of the chosen sensor network solution in terms of transmission delay and message-loss.

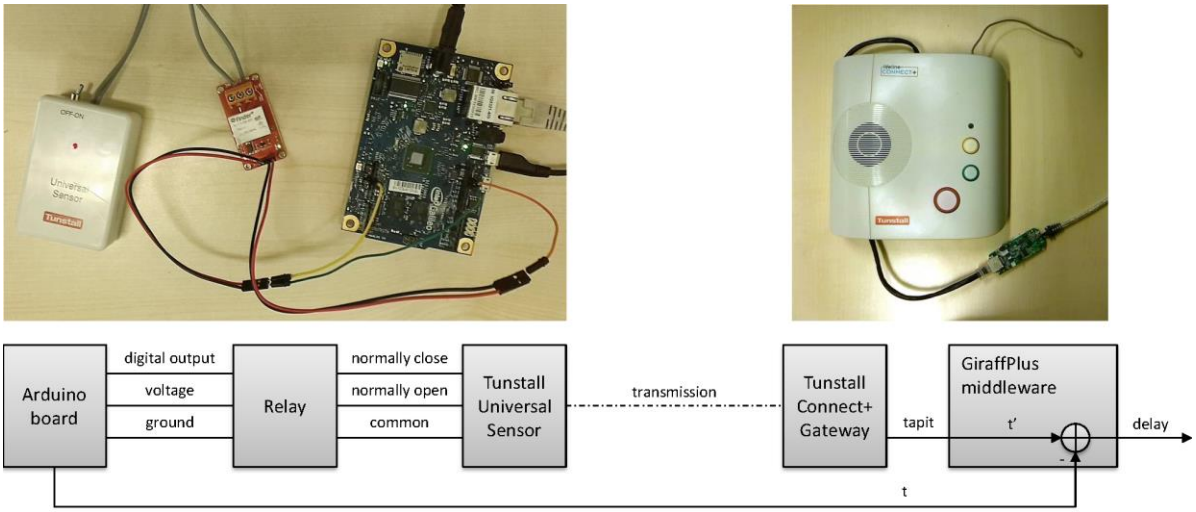


Figure 3 The test chain

Figure 3 shows the laboratory testbed used. It is composed of an Arduino-based [3] Intel® Galileo Development Board controlling a relay that switches on/off the Tunstall Universal Sensor simulating a sequence of 2000 messages for chair pressure on/off at variable interval from 10 to 30 seconds.

We calculated the transmission delay for each message comparing the timestamp *t* generated by the Arduino board sent through the control bus of the middleware and the timestamp *t'* received by the gateway and sent to the context bus of the middleware through the Tapit interface.

Figure 4 shows the probability density function of the obtained results in terms of transmission delay. A very noticeable result is that on all the tests we received all the message sent, obtaining a zero message-loss. It must be noted that this high level of reliability is achieved at the expense of quite high average delay in message delivery. We observed an average delay of 3519 milliseconds with a standard deviation of 713.98 milliseconds. Such times are acceptable for GiraffPlus applications that deal with near real time scenario e.g., long term monitoring of user mobility, time spent in particular locations, and alarms which must be managed by human intervention (typically in the order of minutes).

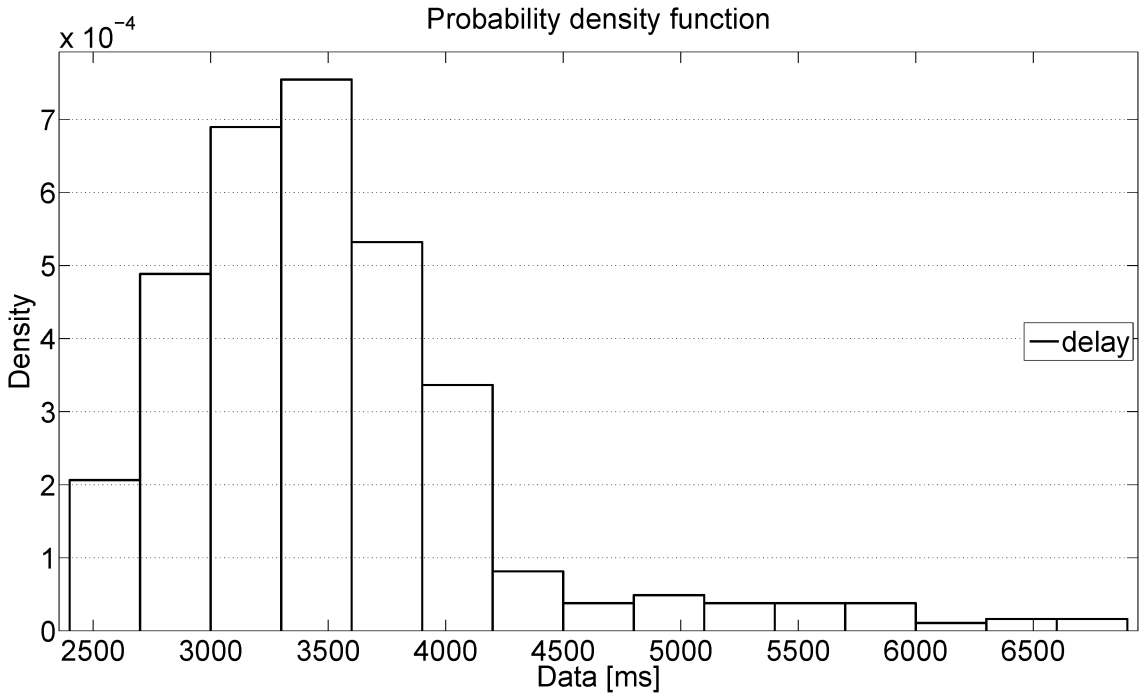


Figure 4 Probability density function of the transmission delay

Figure 5 shows the cumulative distribution function of the transmission delay. We define the error ε as the difference between t and t'. The Cumulative Distribution Function (CDF) of ε is the probability that the transmission delay takes a value less than or equal to e milliseconds and it is defined in equation (1). The obtained CDF shows a 3785 milliseconds tertile (delay less than or equal to 3785 milliseconds in the 75 percent of measurements).

$$F(e) = P(\varepsilon \leq e) \qquad (1)$$

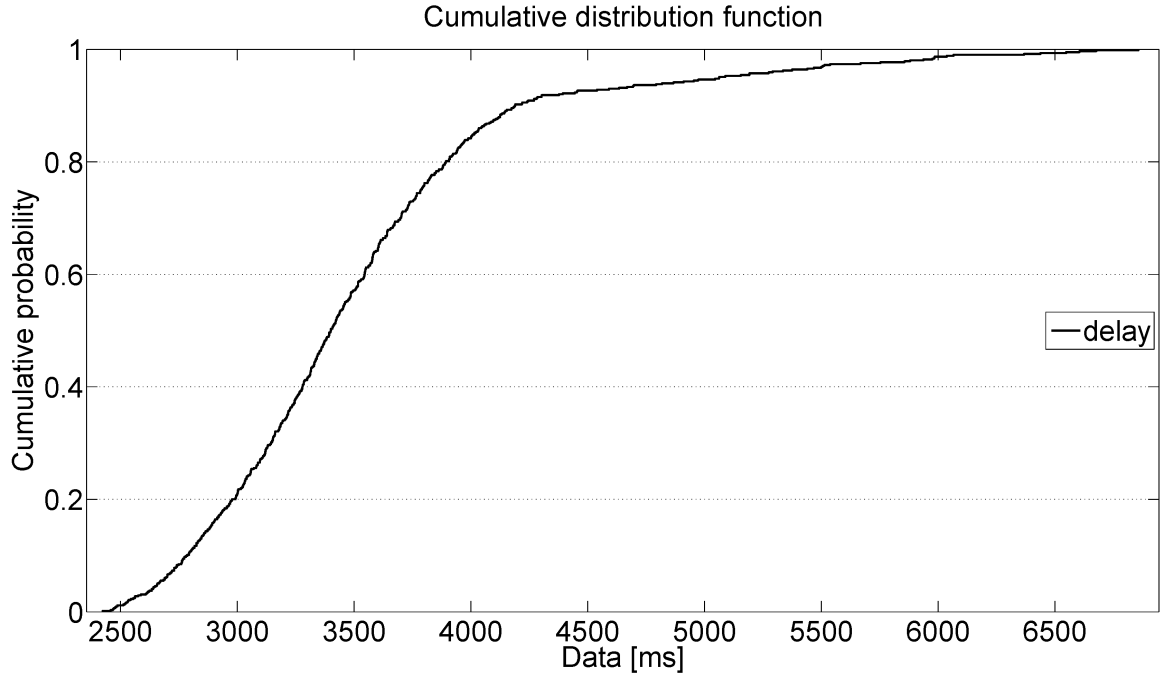


Figure 5 Cumulative distribution function of the transmission delay

These results demonstrate that this kind of solution fits well in a long-term monitoring scenario where the need of reliable data is larger than the need for low latencies in transmissions.

2.3 Additional sensors selection and integration

During the project, the need of collecting more data about the elderly activities arose and to satisfy that, additional sensors have been selected to be integrated into the GiraffPlus platform. These allow to study with more detail the elderly activities and also to generate additional alarms in case of detected abnormal movements like falls.

2.3.1 Wearable inertial sensors

In order to integrate the data coming from the environmental and physiological sensors, wearable inertial sensors can tell when and how much the person is moving and can help in suggesting which activity is performing. One of the ways to achieve this objective is by analysing the wrist movements.

Additional inertial wearable sensors were selected to be integrated with the Android mobile version of the GiraffPlus platform. After an accurate review of the commercial available solutions, the Pebble SmartWatch (Figure 6) was picked as the more appropriate choice for the purpose of the project, in that it features a good trade-off between user acceptability and additional monitoring characteristics:

- Open development platform
- Android API available
- Good battery life (about one week)
- Bluetooth 4 with low energy capabilities
- Accelerometer
- Vibration
- Data logging
- Persistent storage
- Accessible price (around 150€)

Figure 6 The Pebble SmartWatch

Additionally, custom Pebble applications can be implemented in C language and deployed on the watch through a command line tool that could be also used to download the real-time log generated by the system.

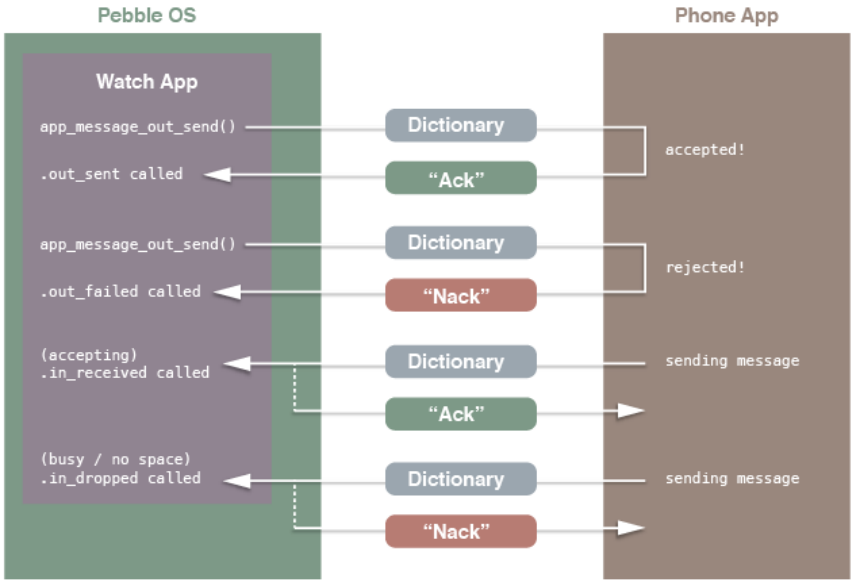


Figure 7 Pebble App and Phone App possible communication diagram

The communications between a Pebble app and an Android app take place by exchanging a set of key-value pairs (called Dictionary) following the scheme shown in Figure 7. The API provided is called *AppMessage* and enables a bi-directional communication between the devices. By using the callback mechanism, the watch app is notified when a message is received from the android phone/tablet and vice versa. Moreover an acknowledgment message allows to be notified in case of errors while sending or receiving data.

The Pebble hardware accelerometer is able to detect taps, perform measures at a given frequency, and transmit samples in batches to save CPU time and processing. In the API, data events on a three-dimensional axis are simply enumerated as raw x, y and z 16-bit signed integers. Each value is measured in milliGs. Pebble accelerometer is calibrated to measure a maximum acceleration of +/- 4G. When looking at Pebble, axes (shown in Figure 8) are standardized to the following coordinates [4]:

- X+ Specifies left to right axis
- Y+ Bottom to top axis
- Z+ Coming up out of the watch

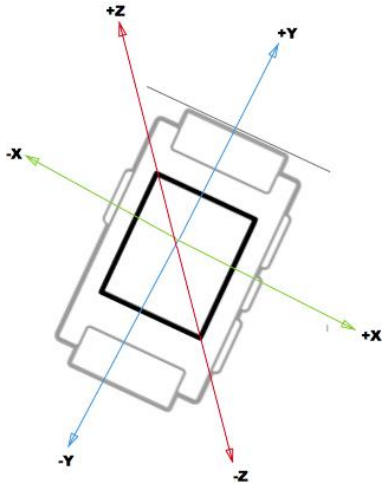


Figure 8 Pebble watch reference axes

On the phone side, an app provided by the Pebble manufacturer is used to communicate with the watch, providing to the developers an easy and efficient interface to interact with the Pebble. This app also allows downloading from the Pebble Store additional apps to personalize the watch functionalities like tracking the running activities, receiving news alerts, calling predefined phone numbers, setting alarms and reminder in line with the GiraffPlus use cases.

The application developed for the GiraffPlus system is an Android service that receives the accelerometer data (x, y and z acceleration values expressed in milliGs) sampled at 10Hz from the watch and sends it to the GiraffPlus system. This data can be elaborated to help detecting falls or understanding the current activity carried out by the person wearing the watch. To receive data from the Pebble app, the Android service implements a special purpose BroadcastReceiver provided by the PebbleKit:

```
private PebbleKit.PebbleDataReceiver pebbleReceiver = new
      PebbleKit.PebbleDataReceiver (PEBBLE_MONITORING_APP_UUID)  {
      @Override
      public void receiveData (final Context context, final int
            transactionId, final PebbleDictionary data) {
            // Elaboration of the received data
      }
```

The Pebble app is started or stopped on demand in order to save energy by calling the correspondent:

```
PebbleKit.startAppOnPebble (context, PEBBLE_MONITORING_APP_UUID)
```

and:

```
PebbleKit.closeAppOnPebble (context, PEBBLE_MONITORING_APP_UUID)
```

Every Pebble App is identified by a unique ID that is defined in a manifest file at compile time and that has to be provided every time we need to interact with that app.

The Pebble SDK is definitely a fast and efficient development environment that reduced the time needed to integrate the device into the GiraffPlus system. Additional applications like sending notifications to the users both visually and with vibration can be developed to increase the interaction level with the assisted persons. Moreover the device is expected to have a good level of user acceptance in that it is not invasive and with size comparable with normal watches. This makes the Pebble watch a good choice for extending the GiraffPlus platform.

Additional services can be developed to increase the feedback level from the primary users through simple questions displayed on the screen and reported back to the DVPIS, tightening the connection between the elderly and the caregivers.

2.3.2 Pulse Oximeter and Fall Detector

Android-based sensor platform has been successfully integrated into the GiraffPlus platform. As a result, MDHPulseOximeter and MDHFallDetector were added to the system (via Engineering GUI) as complementary sensors. Additional control functionalities were introduced, including wireless network availability, average activity check and “switch on/off status” notifications.

A separate application has been developed to perform activity monitoring and fall detection with alarm notification functionally based on Pushover [5] and GiraffPlus middleware. The user is invited to choose the communication option depending on monitoring circumstances.

Subsequently, a number of real-life tests were initiated to perform continuous monitoring of physiological parameters and to assess both entities in terms of user acceptance, data collection and communication reliability. Application has been tested and integrated into the monitoring process.

With latest improvement, complementary sensors are able to communicate with caregivers directly in case fall alarm has been triggered. Subsequently, a number of real-life tests were initiated to perform continuous monitoring of physiological parameters and to assess both entities in terms of user acceptance, data collection and communication reliability. Both sensor functionalities were involved into the current research towards context aware fall risk assessment, resulting in a journal publication [6].

3 The Middleware

In addition to the smooth integration of equipment for sensing the environment, a middleware solution is necessary in order to support the interaction between smart services and user interfaces defined and deployed in the system. This middleware solution described in its main functionalities in D2.2 has proven useful hiding heterogeneity and distribution of both hardware and software resources.

The final version of the middleware has been delivered al month 30 and presented in [7], featuring new functionalities and hardware and operating system compatibilities. These new features will be described in details in the following subsections together with the tests made in order to guarantee robustness and scalability of the middleware.

3.1 The desktop and mobile stable versions

Stable versions of the mobile and desktop middleware have been developed and deployed in the running test sites. The code is available with ASL license [8] on the GiraffPlus svn (https://giraffplus.xlab.si/svn/giraff/trunk), the mobile middleware service is available for installation to GiraffPlus professionals on the Google play store [9], and the procedures and best practices useful to install and maintain the middleware components are available on the GiraffPlus wiki pages (http://wnlab.isti.cnr.it/giraffplus/).

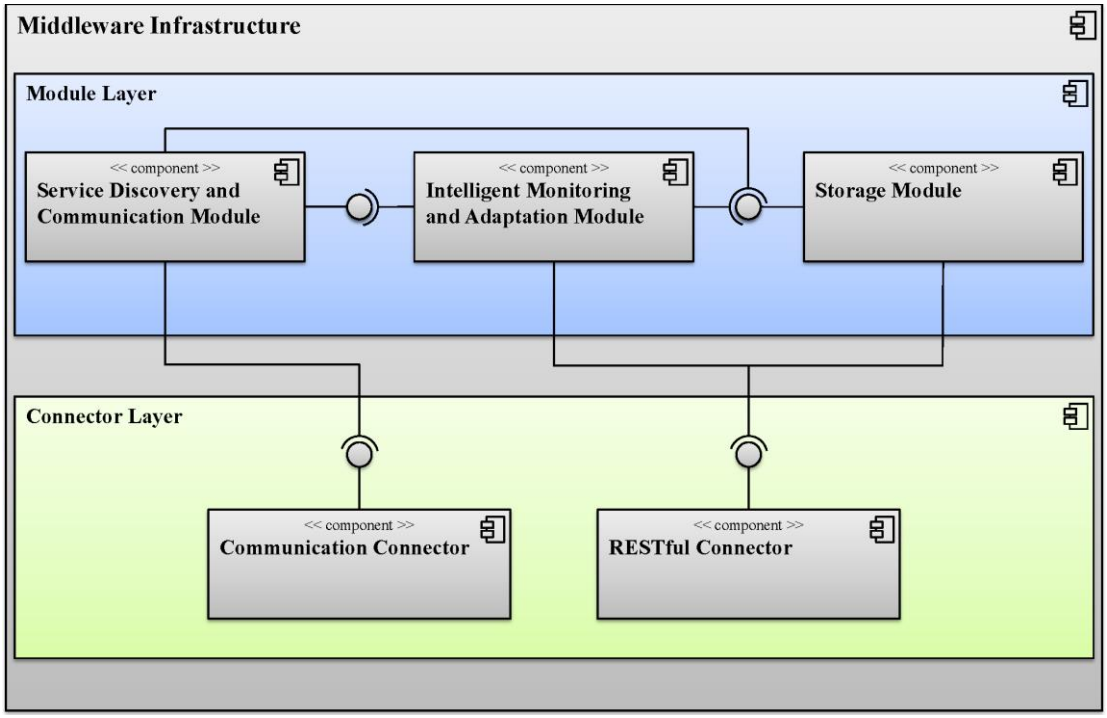


Figure 9 An in-depth view of the middleware component

Figure 9 shows the component diagram of the final middleware desktop version reflecting the reference architecture described in D2.2, while Figure 10 shows the component diagram and class diagram of the final android middleware.

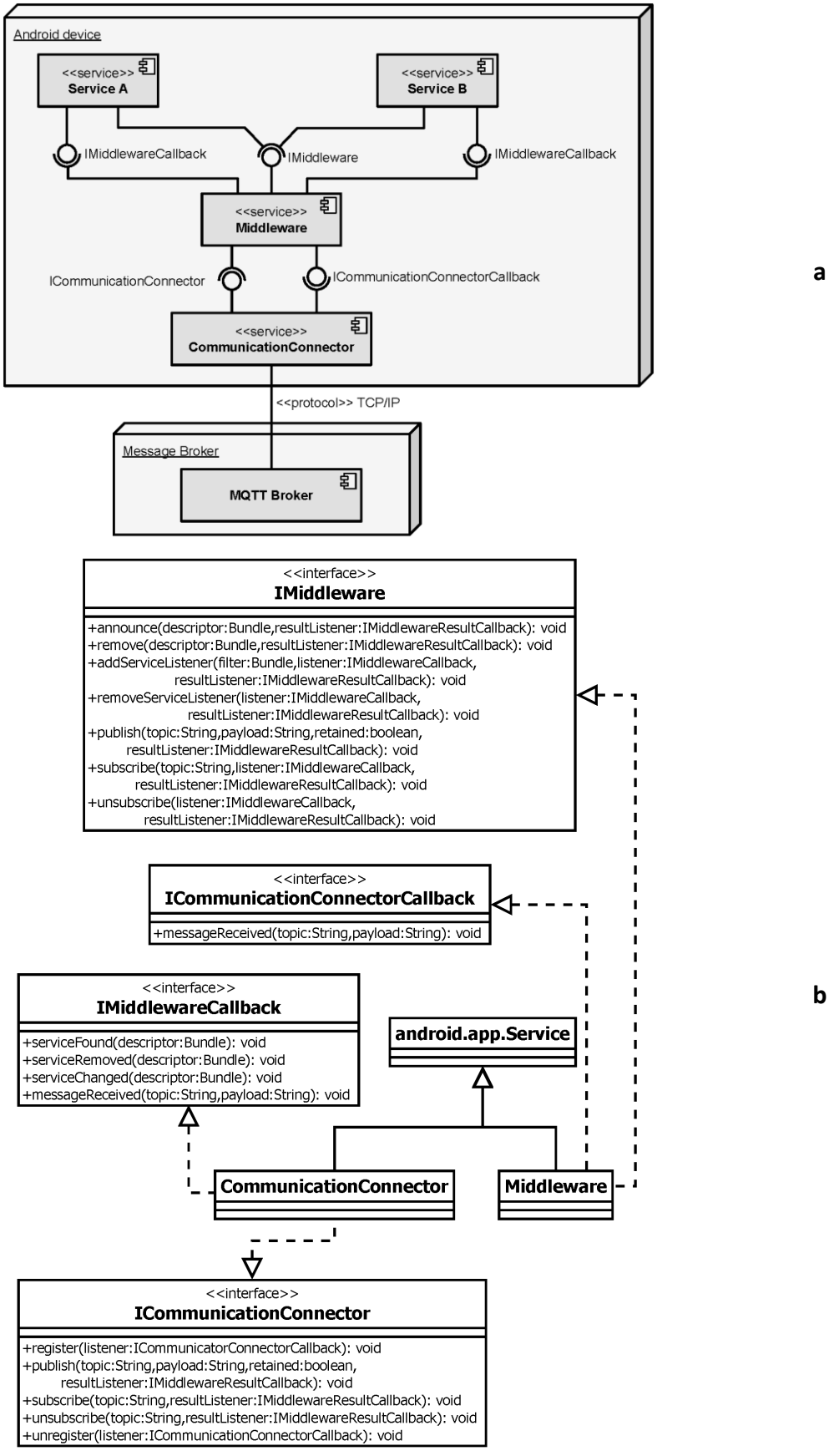


Figure 10 The Android middleware architecture with component (a) and class (b) diagram

3.2 New features

Some new features were added to the GiraffPlus platform, mainly to improve the reliability of the overall system. The additional features with respect to the deliverable D2.2 are presented in the following sections.

3.2.1 Enriched interfaces

In order to address the continuous feedback from developers and to accomplish the requirement of having a feedback about the status of the middleware instances deployed in the test sites, concerning the desktop version of the system, the Middleware interface has been extended with the following functionalities:

- `public Set<String> getLocation ()`
  This method returns the set of locations to which the current instance of the middleware has access, based on the certificate used to connect to the server

- `public boolean isReady ()`
  This method returns whether the middleware is ready to be used and it should be called before any other method of this class

- `public boolean configure (Dictionary properties)`
  This method allow the reconfiguration of the middleware instance at run-time passing a set of key-value pairs containing the configuration information

- `public LocationStatus getLastHeartbeat (String location, String instance)`
  This method returns the information about the last heartbeat message received from all the locations to which this instance has access. This method is used only while the middleware is in “office” configuration

The CommunicationConnector interface has been also modified as well to comply with the changes of the Middleware component with the following new functions:

- `public int getPendingOperations ()`
  This method returns the number of operations that are currently pending to be carried out by the MQTT library. Any reconfiguration of the connector should verify that there are no pending operations before continuing

- `public boolean configure (Dictionary properties)`
  Like in the previous interface, this method allow to reconfigure the communication connector at run-time

- `public void registerListener (CommunicationConnectorListener listener)`
  This method allows to register a callback to be notified when the connection with the MQTT broker is lost or established

Regarding the Android version, the following methods were added to the IMiddleware interface facilitating the process of retrieving the location configuration settings:

- `String getLocation ()`
  Similar to the desktop version, this method returns the location configured within the middleware instance

- `String getAllSensors ()`
  This utility method returns the list of the sensors installed in the configured location as a JSON formatted string.

- `String getSensorsByManufacturer (String manufacturer)`
  This utility method returns the list of the sensors like the previous one but filtered by a manufacturer name.

Also the GiraffPlus Storage API underwent changes to reflect modifications in the data schema of underlying data entities and in the authentication mechanism (see deliverable D5.4).

As before, the GiraffPlus Storage API allows other components to save, update, delete or retrieve all entities defined in the changed data schema. There are, however, some specifics connected to the data schema – e.g., a sensor entity cannot be retrieved individually, as it is embedded in the home configuration entity, thus one must retrieve the whole home configuration entity and then select the desired sensor within that entity.

Whereas this may seem a very limiting factor, we must understand that these entities are mostly retrieved by the DVPIS or individual middleware instances, which require the entire home configuration entity in any case – e.g., the DVPIS, to correctly visualize the home configuration and retrieve sensor measurements, and middleware instances, to correctly initialize all components and sensors in the home.

In terms of authentication and authorization, the GiraffPlus Storage API must now be provided with a valid GiraffPlus certificate identifying the user or component trying to access data via the Storage API. This certificate is then used to authorize and authenticate against the GiraffPlus Storage LTS Web Service and to encrypt all communication between the Storage API and the web service.

Two new functions were added to the GiraffPlus Storage API:

- `List<SensorData> sendQueryWithPreSample (int numOfThreads, ObjectId sensor, Date start, Date end)`

  The function returns, in addition to sensor readings in the defined time interval, the last sensor reading before the defined time interval. This function is used in the DVPIS and enables better visualization of certain kinds of data.

- `Hashtable<ObjectId, List<SensorData>> sendMultiQuery (List<SensorQuery> queries)`

This function allows the user to simultaneously define multiple queries (e.g., when initializing the view in the DVPIS), which is faster than making multiple queries due to latencies introduced by the authentication and authorization mechanisms.

3.2.2 Heartbeat

A feature strongly requested from the continuous evaluation and integration of the system in the test sites was the possibility of monitoring the current state of the software modules running on the test sites to notifying the technicians in case of software issues. Therefore, in order to allow a real-time surveillance of the test site status, a heartbeat mechanism was introduced. The heartbeat is basically a message sent periodically (currently every minute) on the control bus that signal the middleware presence and gives useful information to the technician in charge of monitoring the test sites. The topic on which this message is published is:

```
<location>/controlBus/heartbeat/<instance>
```

where <*instance*> can be: *home*, *tablet* of *giraff*. Depending on the device on which the GiraffPlus middleware is running, different information are sent within the heartbeat message. This message is "retained", that means the server will store and publish it to the clients even if the sender goes offline. This allows receiving the last status in case a middleware instance crashes or goes offline.

For the *home* and *giraff* versions the heartbeat contains:

- *date*: a timestamp of the sender time
- *uptime*: the time elapsed since the middleware has been started
- *mw-version*: the version of the middleware bundle currently installed
- *mqtt-version*: the version of the mqtt connector bundle currently installed
- *local-db*: a flag that indicates wheter the local db backup is enabled or not
- *local-address*: local IP addres of the machine on which the middleware is running

while for the *tablet* mobile version the heartbeat contains:

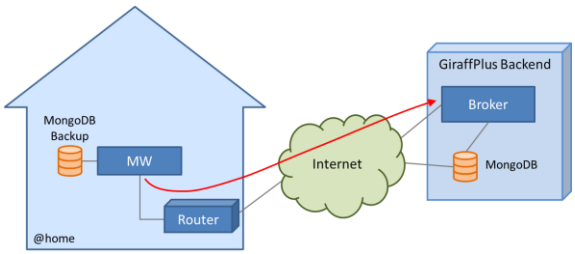
- *date*: a timestamp of the sender time
- *uptime*: the time elapsed since the middleware has been started
- *imei*: the unique identifier of the device
- *plugged*: whether the mobile device is plugged in to the mains or not
- *battery-status*: the state of the battery (charging, discharging, full, not charging)
- *battery-level*: the currently level of the battery (from 0.0 to 1.0)
- *mw-version*: the version of the middleware app installed
- *mw-install*: the installation date of the middleware app
- *mw-updated*: the last update date of the middleware app
- *mqtt-version*: the version of the communicator connection app installed
- *mqtt-install*: the installation date of the communicator connection app
- *mqtt-updated*: the last update date of the communicator connection app

- *onecare-version*: the version of the OneCare app installed
- *onecare-install*: the installation date of the OneCare app
- *onecare-updated*: the last update date of the OneCare app

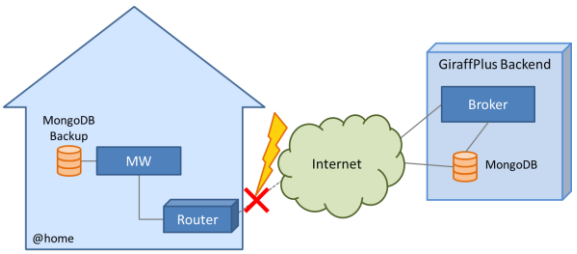
When a middleware instance is started with an "office" configuration, it keeps track of all the heartbeats received from the "home", "giraff", or "tablet" instances to which it has access, and by a call to the previously described `getLastHeartbeat()` the correspondent status can be retrieved.

3.2.3 Failover mechanism

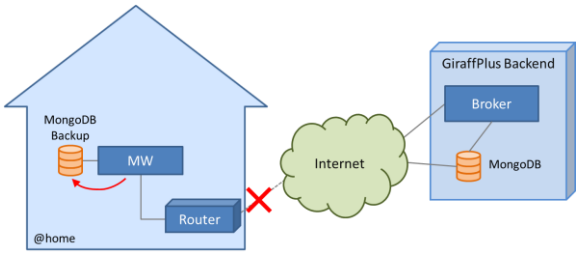
Another important request arose during the deployment phase of the system was a long term backup mechanism. The development of this feature was triggered by the loss of data in some test sites after an episode of long network disconnection. To improve the reliability of the system in case of long term connection outages, a failover strategy was implemented through the usage of a local instance of MongoDB [10]. When the system detects a connection problem it simply switches to a local installed database and starts to save the data within it. Later on, when the network connection is restored, then the locally saved data is sent to the remote server (Figure 11).



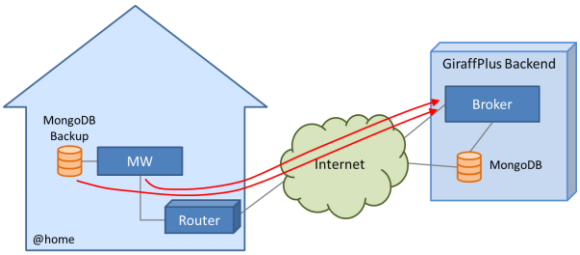
Phase 1. Network is working normally



Phase 2. Network goes down



Phase 3. Messages are sent to the local Mongo database



Phase 4. Network connection is established again and the data is sent to the remote server

Figure 11 The failover mechanism phases

The module that provides an interface with the local database is MqttBackup2Mongo. The main methods exposed by this module are the following:

- `public void saveMessage(String topic, String message, boolean retained)`
  This method saves the message on the DB
- `public List<Object[]> getSavedMessages()`
  This method returns the list of the messages present on the local DB. Every entry of the returned list is an array composed of four fields <*topic*, *message*, *retained, id*>
- `public void dropMessage (String id)`
  This method removes from the database the entry with the given id

Going into the details, an example of typical interaction during a network outage is shown in Figure 12.

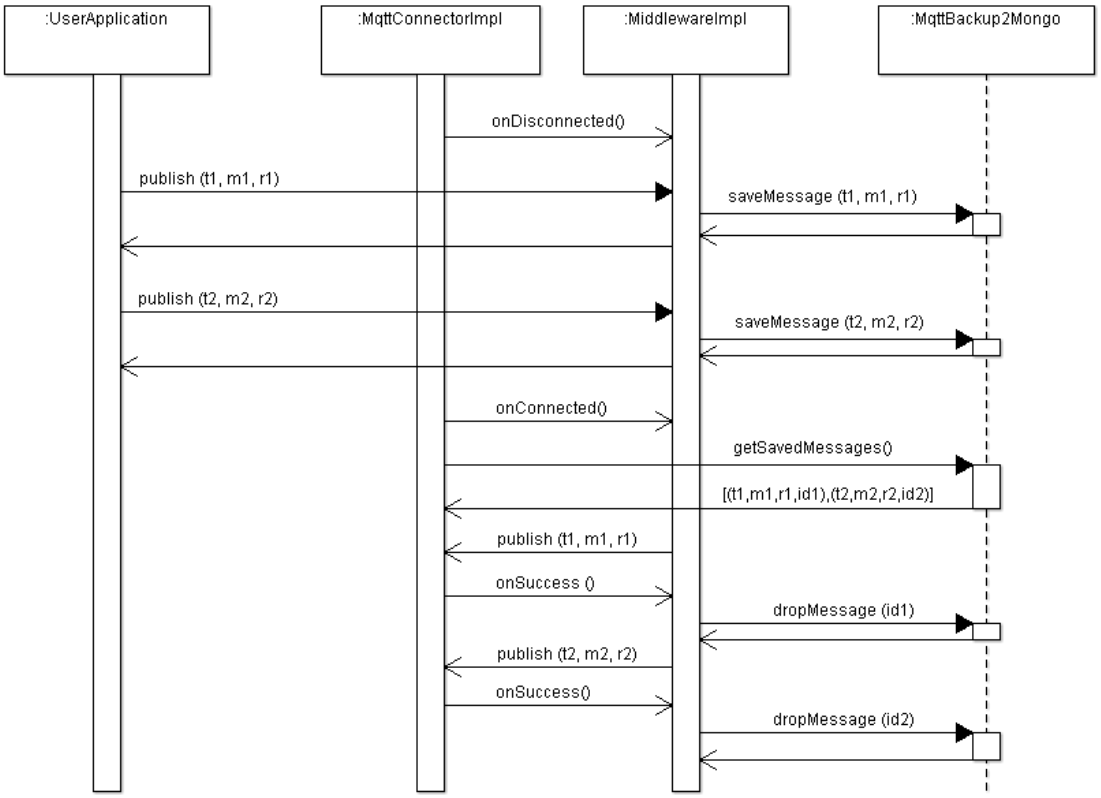


Figure 12 The failover mechanism sequence diagram. In the figure t_i represents i-th the topic, m_i represents the message, r_i the retained flag for the i-th message, and id_i the i-th identifier assigned by mongo db.

The MongoDB is installed as a Windows service and configured with automatic startup during the testsite deployment. No further actions are required for its normal operation.

3.2.4 Dynamic reconfiguration

To improve the system flexibility, the possibility to programmatically change the middleware configuration without the need to restart the related bundles was added. Is it now possible to call the method configure() on the middleware instance passing the following properties (with example values) as a Dictionary:

- mw.keystore.trust.pwd=*******
- mw.keystore.trust.path=..//trust.jks
- mw.keystore.client.pwd=********
- mw.keystore.client.path=..//client.jks
- mw.configuration=home
- connector.keystore.trust.pwd=*******
- connector.keystore.trust.path=..//trust.jks
- connector.keystore.client.pwd=********
- connector.keystore.client.path=..//client.jks
- connector.host=giraffplus.xlab.si
- connector.port=8883

Whenever that method is called with a new configuration, the middleware takes care of unsubscribing from all the subscribed services, cleaning the internal state and automatically configuring the communication connector module as well. In case a property is missing, the configuration procedure is aborted and signaled to the calling application.

3.3 Guaranteeing the middleware robustness

A constant performance and resiliency assessment is necessary when the software is deployed in the real test sites to guarantee the correct behaviour during unexpected situations like bugs in the code, reaction to network misconfigurations, updates of third party software, etc.

3.3.1 Maintenance and bug fixing

A continuous maintenance activity was carried out during this period of the project and, since the deliverable D2.2, some bug were fixed in both the desktop and mobile version of the GiraffPlus middleware.

An issue causing the impossibility to reconnect to the MQTT broker after a connection drop was fixed by properly regenerating the SSL context before a new attempt to reconnect.

A limitation regarding the certificate keystore version was preventing the Android middleware to open a BKS keystore with a version greater than 1.46. This because Android uses a customized version of the library that cannot be updated. A workaround was found by using the SpongyCastle library [11], a third-party project that can handle any version of the keystore.

Some checks and verifications were added in the code to ensure a proper behavior of the system even in case of uncommon combination of conditions that could have led to the instability of the system.

A real-time monitoring application, exploiting the heartbeat functionality, was developed in order to get an overview of the working status of all the test sites. A web interface that will be integrated in the EngineeringGUI is planned to be implemented.

3.3.2 Scalability

In order to evaluate the middleware performances, we run several tests on an experimental platform composed of a GiraffPlus middleware instance on a desktop computer running also a MQTT broker and two instances of mobile middleware running on smartphones equipped with an ARM Cortex A8 1.2GHz processor and Android 4.0.4. We measured the performances of GiraffPlus mobile middleware in terms of latency introduced by the middleware to manage the publish requests made by a producer service and to dispatch messages to a consumer service. In particular, we aim at observing how latency scales with the number of requests per second (rps) and the number of consumers and producers concurrently running on the same device. These tests were deliberately chosen as an extreme conditions scenario in order to test GiraffPlus dealing with a very high number of services simultaneously running on the same node.

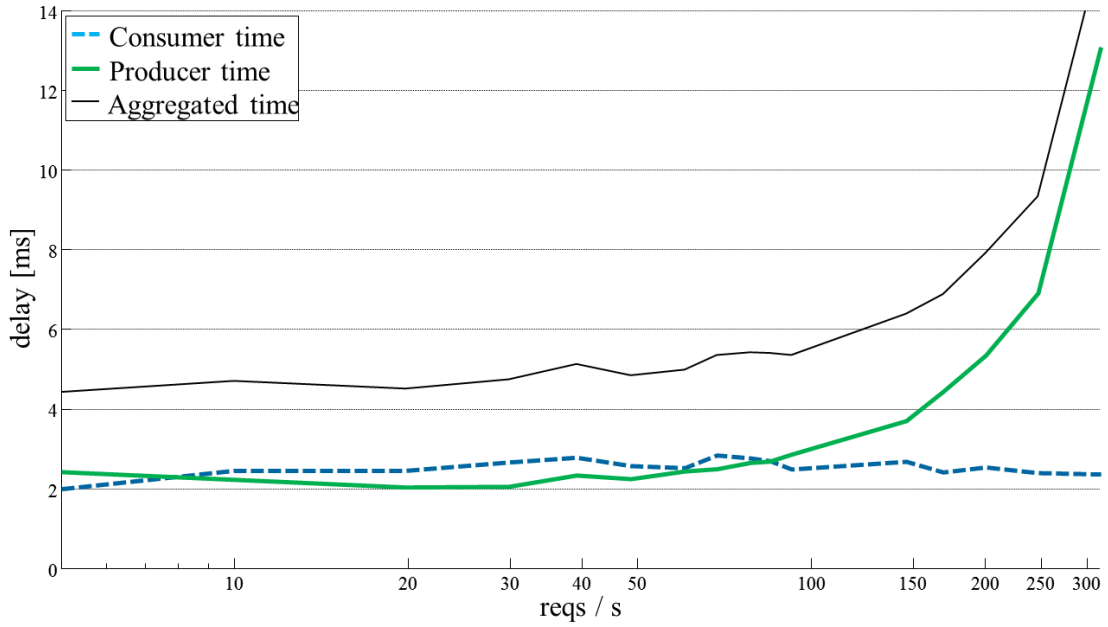


Figure 13 Middleware latency with 1 producer and 1 consumer varying the requests per second

Figure 13 shows the time (in milliseconds) spent by the middleware to accomplish the publish request of a producer sending a message of 100 bytes to the network (Producer time) and to dispatch the message to a consumer once it is received from the network layer (Consumer time). We observed that the aggregated time remains in the range of 5 ± 0.5ms from 1 to 100rps and grows up to 14ms at 300rps. After that value the mobile middleware stops handling the requests returning a *TransactionTooLargeException*. This behavior is strictly connected to the hardware limit of the tested device: the GiraffPlus mobile middleware process cannot handle such a high number of requests per second that saturate the Binder transaction buffer (that has a limited fixed size of 1 Mb and it is shared by all transactions in progress for the process). This is reasonable especially because in AAL

application the system should react with timings compatible with the user, which hardly requires high frequencies [12]. It should be note however that, if higher frequency in data sampling is required, the actual rate of communication is generally lower due to the need for data aggregation and fusion techniques [13].

In further tests on scalability we have considered frequencies up to 5rps for a producer service. Figure 14 shows the scalability of the GiraffPlus mobile middleware in terms of number of concurrent producers on the same device. We identified a limit of 40 producers at 5rps with 100B of message payload. Also in this case the limitation is due to the hardware of the tested device for the same reason of the previous case.

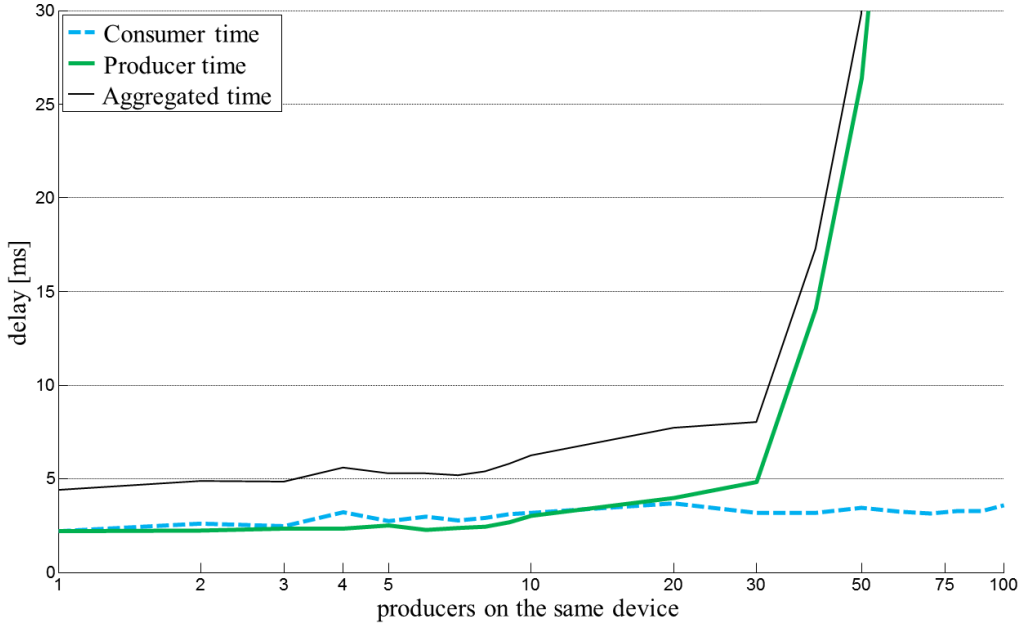


Figure 14 Middleware latency with 1 consumer varying the number of producers transmitting at 5 requests per seconds

Finally, Figure 15 shows the scalability of GiraffPlus mobile middleware in terms of number of concurrent consumers on the same device varying the number of concurrent producers transmitting at 5rps with 100B of message payload. Each consumer is subscribed to all the present producers in the network. The middleware can handle, with an aggregated latency under 30ms, up to 10 concurrent consumers in presence of 50 producers, up to 20 consumers when 25 producers are transmitting simultaneously, up to 30 consumers with 10 concurrent producers, and more than 100 consumers subscribed to a single producer.

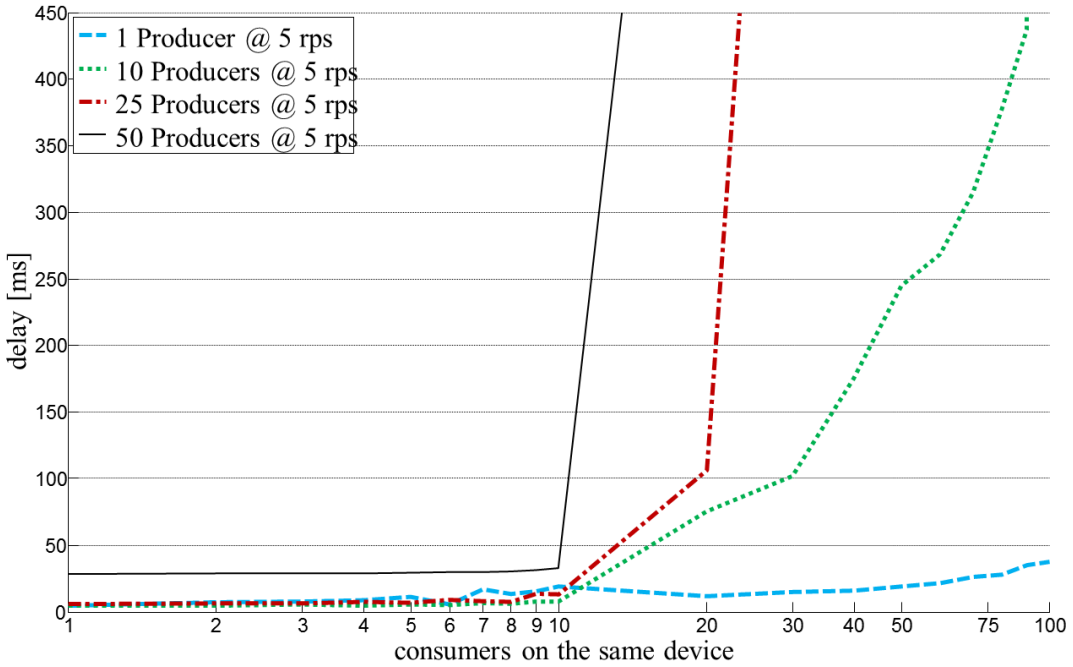


Figure 15 Middleware latency varying the number of consumers with 1, 10, 25, and 50 producers transmitting at 5 requests per seconds

We tested the GiraffPlus middleware in particular stress conditions with the mobile device acting as a single aggregation point of all the possible services installed in the home environment.

We also tested the context-awareness of the GiraffPlus mobile middleware developing a concrete application that turn off unnecessary network adapters like bluetooth or WiFi once it detects an outside scenario. We believe that such an application is very useful to limit the energy consumption of mobile devices exploiting the context data coming from the GiraffPlus network. An elderly person living alone in her house is a scenario particularly dear to the AAL community. Also in the GiraffPlus test sites, elderly people live alone, so when no presence sensor data is sent on the context buses, we can infer that the user is outside the house. To estimate the increased battery saving of this solution, we compared the power consumption of the context-aware application built upon the GiraffPlus middleware with a generic application that periodically scans the WiFi signal strength to fingerprint the home WLAN. We also compared our solution with a generic situation where the user forgets all the used network interfaces on when he goes outside (ALL-ON).

Table 2 Energy consumption in mW

WiFi		Bluetooth			Radio	
on	scan	active	on	active	on	active
5.1	192.7	229.7	2.59	32	7.95	686

Table 2 shows the energy consumption of the network adapters present in our test device taken from its power profile file. In the proposed solution, the GiraffPlus mobile

middleware sends a heartbeat of 100B each 30s using the 3G radio. We estimated the active state of the radio while transmitting the heartbeat during 0.13ms (with an uplink bandwidth of 5.76Mbps). During the duration of the tests (1 hour) the radio was in on mode, so the total radio consumption was 120 × 0.686W × 0.00013s + 3600s × 0.00795W = 28.63J. The WiFi scan application calls a *wifi.scan* with the same period (30s), the scan operation last 500ms in which the adapter is in the active state, and the WiFi was on during the duration of the test, so the total consumption of the WiFi adapter was 120 × 0.422W × 0.5s + 3600s × 0.0051W = 43.68J. We measured the CPU consumption of GiraffPlus mobile middleware and the WiFi scan application using the PowerTutor [14] model. In the ALL-ON scenario, we left the default WiFi scan period of 15s, and we estimated the additional consumption of the Bluetooth interface left on and undiscoverable in 3600s × 0.00259W = 9.32J.

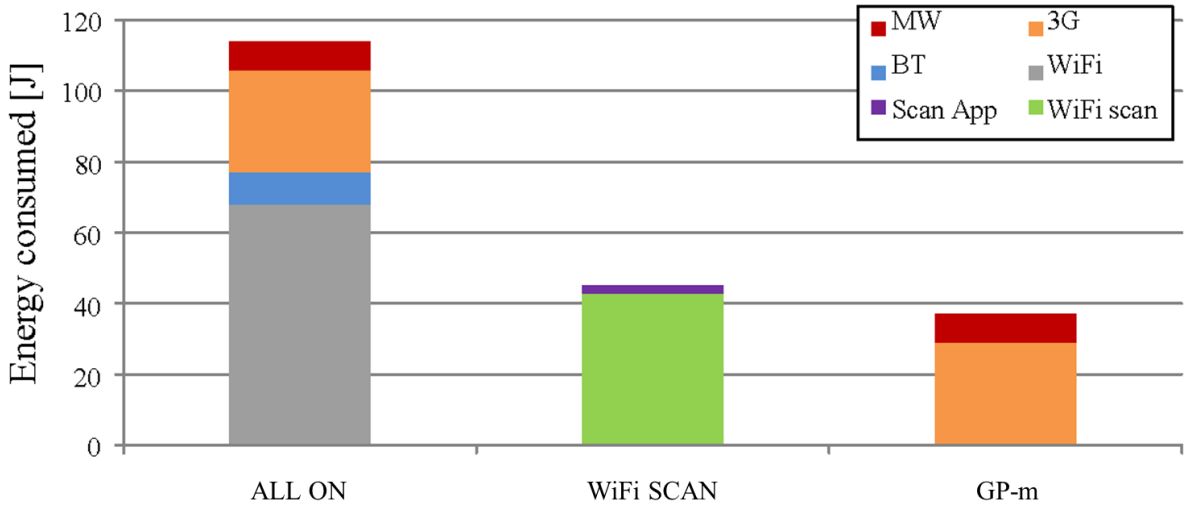


Figure 16 Comparison of energy consumptions in one hour of test of the two different approaches analyzed with respect to the ALL-ON situation. The WiFi Scan saves 60.5% while the proposed solution saves the 67.5% still remaining connected to the GiraffPlus system

In Figure 16 the overall results are shown. Using the context information provided by the GiraffPlus middleware, a simple application can optimize the power consumption of the device remaining connected to the GiraffPlus system and reachable from her caregivers.

The energy optimization represents a crucial aspect in the deployment of the GiraffPlus system on mobile nodes, for this reason further work will be performed in the next period of work of the project.

4 The Giraff Platform

In this section the second and final version of the Giraff platform is presented focusing on the new features implemented. New Giraff software has been developed and released in order to allow collaborative control between the human teleoperator and the onboard motion (4.1) together with new functionalities offered by the platform (4.2).

4.1 Collaborative control

According to the requirements for the third prototype of the GiraffPlus system at month 30, the robotic platform has been enhanced with a collaborative control system, which consists of a high-level controller that extends the capabilities of the prototype presented in deliverable D2.2. It complements the motion commands from the driver with the semiautonomous navigation system to safely pass through doors, corridors, and cluttered spaces.

During this period, a new software module has been implemented to enable a hybrid operating mode with the aforementioned collaborative control feature considering the obstacles detected by the sensing system. The result is a new teleoperation mode where the driving payload is highly reduced while the driver still feels s/he is controlling the robot.

a **b**

Figure 17 Two typical cases where the collaborative control proves its suitability for teleoperation. a) The user drives the robot towards an obstacle. b) The user wants to cross a door by marking a point inside the room. In both cases the collaborative control generates appropriate paths to arrive the user destination while negotiating obstacles.

Figure 17 shows two typical scenarios where the collaborative control is useful. Figure 17.a shows how the driver selects a trajectory colliding an obstacle (red line). In this situation the collaborative control intercepts the user intentions and generates a free-obstacle path. In Figure 17.b the driver drives the robot through a door by simply selecting a point inside the room without paying attention to the narrow space of the door. The collaborative control identifies the user wants to go into the room and generates the proper path negotiating the door frame.

The Collaborative Control has been integrated into the secondary user application, i.e. Pilot. Specifically, it has been integrated into the semiautonomy plugin presented in D2.2.

Figure 18 shows a screenshot of the current version of the user application, highlighting the collaborative control option. The driver can select the collaborative control by selecting the option into the left panel. A text message as well as a different color scheme for the application indicates that the feature is active. Under this mode the driver commands the robot as usual although her/his commands are (can be) override by the semiautonomous navigation system to negotiate obstacles.

Figure 18 Secondary User Interface in Collaborative Control Mode

Figure 19 shows the robotic architecture including the collaborative control module and its relations with the rest of modules. It captures the destination marked by the driver and generates a target which is inputted to the reactive navigator of the robot. So, the collaborative control can be seen as an intermediate layer between the driver and the semiautonomous navigation system presented in D2.2.

The collaborative control module considers three security levels for which a behavior is selected for the robot navigation:

- ***Level 0*: Obstacles-free navigation mode.** No obstacles are detected nearer than 1 m. and the collaborative control system does not override the user commands.

- ***Level 1*: Obstacles sensed in the range 1m to 4 cm.** User navigation is supervised by the semi-autonomy system, i.e. the user commands are transferred to the reactive navigation algorithm which generates a secure path.

- ***Level 2*: Maximum risk; obstacles under 4 cm.** In this case, the semi-autonomy system commands an emergency stop and limits the movements of the robot to avoid collision. In this mode, the robot can only rotate to escape from a dead-path.

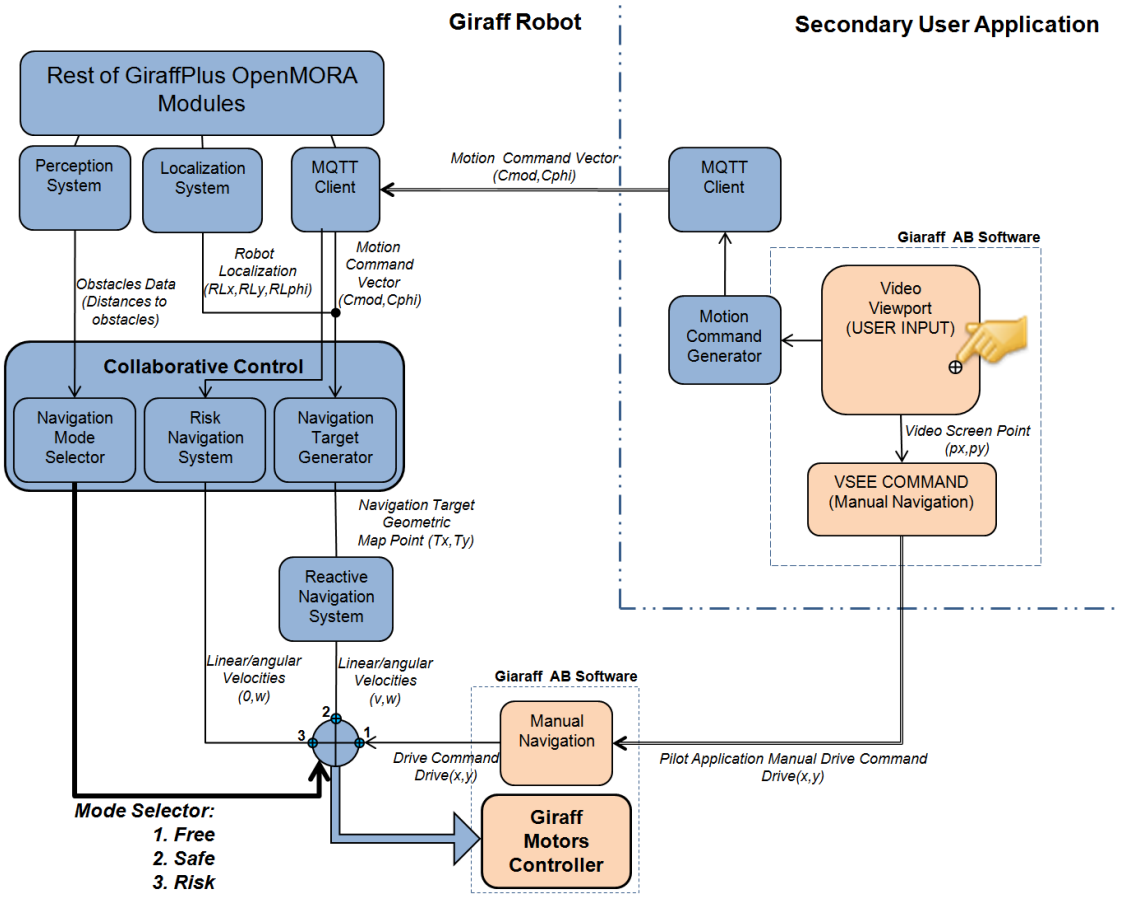


Figure 19 Collaborative control integrated with the architecture presented in D2.2

The work done by the collaborative control system is almost transparent for the driver, who commands the robot by clicking on the remote video image received from the robot, i.e. as done for the normal driving option. However, each movement command is appropriated manipulated to generate a target destination suitable for the reactive navigation system. Concretely the destination marked by the driver *(px, py)* is captured from the viewport in polar coordinates and transformed from the local reference frame of the image to the global localization frame used by the semi-autonomous navigation system (Figure 20).

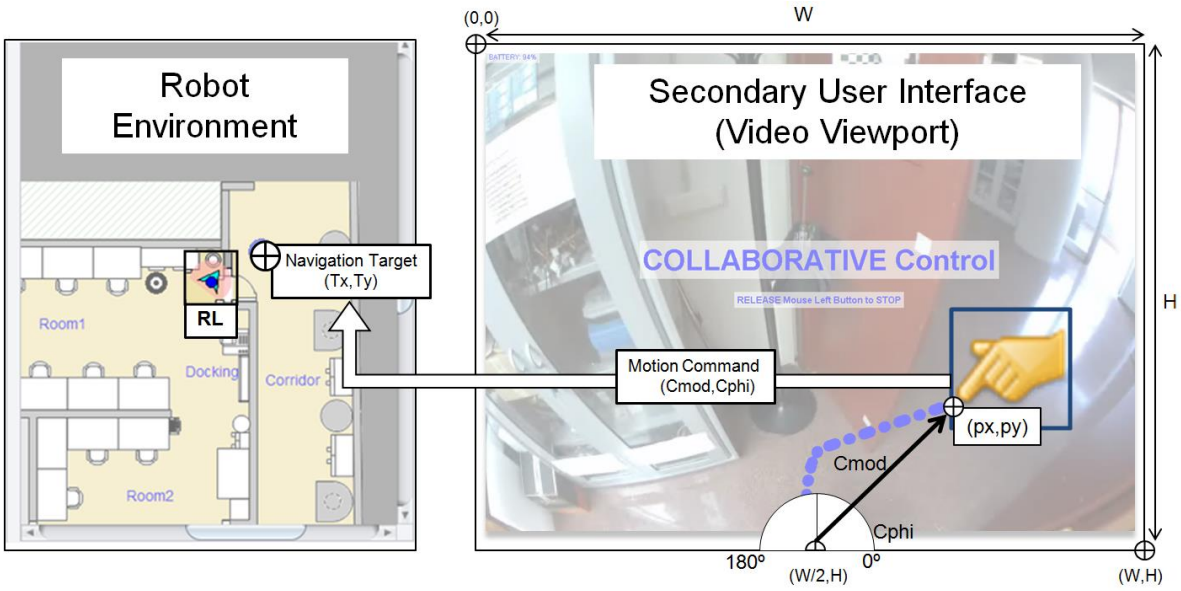


Figure 20 Selection of the user destination and transformation into the global reference frame of the localization system

In summary, the collaborative control algorithm is as follows:

1. The system receives a user command in terms of the module and angle of the selected destination *(Cmod,Cphi)*.

2. The security level is decided according to the obstacles sensed.

    a. For low risky situations (obstacles farther than 1m) the user commands are directly sent to the motor system.

    b. For risky situations (obstacles between 1 m. and 4 cm.) a target for the reactive navigator is composed by the robot pose *(RLx,RLy,RLphi)* given by the localization system and the polar coordinates of the destination *(Cmod,Cphi)*.

    c. For highly risky situations (obstacle under 4 cm.) the collaborative control only permits rotations setting the angular speed *w=sign(Cphi)*Wmax*Cmod*

3. Go to step 1 while the collaborative mode is switched on.

4.2 Giraff software

The new software version (v2.4) for both Giraff and Pilot developed and released during this period includes new features for the 4.0 version Giraffs which are related with the night vision and height adjustment.

The feature of sending an alert when a Giraff is not charging has been developed.

Additionally, the new software includes a number of bug fixes and improvements that increases the overall reliability.

The new software includes a lot of release testing. Moreover, the latest software version contains a few bug fixes and support for the new remote control. An in depth overview of the changes and bug fixing performed is given in Appendix 1 - Selection of changes and bugfixes to the Giraff and Pilot software of release 2.4.

Appendix 1 - Selection of changes and bugfixes to the Giraff and Pilot software of release 2.4

- Assisted docking drive plugin no longer sticks until next call. If you hang up with this plugin active the default drive plugin will be active when the next call starts.
- Added Giraff User's Guide.
- Added Vsee configuration files for the 4.0 Giraff.
- Updated updateVSeeConfig ruby script to work for 4.0 Giraff.
- Major rewrite of the virtual camera driver for the camera present on the 4.0 Giraff.
- Added support for starting Giraff with OpenMORA/AVR.
- Created an API for closing the Sentech camera manually.
- Added a grace period before killing Vsee, allowing it to terminate by itself.
- Added magnification functionality.
- AutoUpdater now terminates the software correctly.
- Increased the polling frequency of the battery status.
- Fixed E28 bugs and made corrections in PID regulator values to work with the hardware failure E41 error for 4.0 Giraffs which has new motors.
- Added night vision functionality.
- Fixed bug causing loss of colour settings when zooming in.
- Changed timeout length for answering calls.
- Refactoring of the virtual camera code, added timestamps to images, sleep mode when not in call.
- Updated battery software version.
- Fixed a bug in the assisted docking plugin causing the algorithm to select a worse target candidate over a better one.
- Fixed a bug in the assisted docking plugin causing the bottom of the target rectangle to be used instead of its centre.
- New feature that should protect the tilt motor gear box if someone physically pulls the screen.
- Added support for compiling .elf file containing main program, fuses, bootloader (controller only) and eeprom (charger 2.0 only).
- Minor parameter tweaks for the auto docking algorithm.
- Removed the old (and never used) zoom command and all references to it.
- Refactoring of system errors and changed behaviour to listening for status from the microcontroller instead of inferring errors from other parameters.
- Prepared for using timestamps in 4.0. Now works in all resolutions. Disabled as it does not work with older (<4.0) Giraffs.
- Removed timestamp extraction from VSeeImageGenerator dll. This is handled in java code from now on.
- Changed HTTP API access to use preemptive authentication (saving us one server call).
- Updated Apache HttpClient to version 4.2.5.
- Changed default resolution/framerate to 480p@30fps .
- Added logic in InCallView for handling night vision mode.
- More fail-safe handling of the robotnum configuration file.
- Corrected behaviour of the B indicator on the Giraff.

- Compacted Pilot UI to fit 768 pixel high windows with a menubar.
- Type refactoring of AVR communication layer.
- Refactoring of call state in ApplicationManager.
- Fixed bug that caused some AVR data not present in Giraffs older than 4.0 to be polled.
- Auto-raise neck when hanging up call, warning on Pilot when hanging up if not in raised state.
- Added a top voltage cut-off when charging (another E28 bugfix).
- New extension point (BAT file execution) when starting/quitting Giraff application.
- Fixed a bug causing the Giraff to get stuck when turning it on.
- Completed resource cleanup (network sockets, file descriptors, streams, etc).
- Fixed the bug were the SPI bus gets out of sync when turning off the big microcontroller in the middle of a message.
- No keep-alive for sentry fake ping connection (to avoid new threads being created).
- Giraff will now notify Sentry when left outside charging station.
- Fixed some Swing repainting logic.
- Changed the UI logic for making sure there is no zoom or resolution changes while in night vision mode.
- Correctly sync microphone and speaker volume when loading settings from previous call.
- Changed the error handling when setting up serial port communication.
- Corrected error message shown when logging in without providing a username.
- Added logging to last_error.txt
- Fixed a bug causing exception handling to fail when not logged in.
- Fixed problems related to default resolution in local mode.
- Added a command line switch for full screen mode.
- Fixed a bug causing the network card MAC address not being set in the application model.
- Added a full screen mode to PilotMainWindow.
- Updated translation files.
- Removed Finnish and Chinese languages from Pilot application.
- Fixed a UI bug that showed up when switching from 720p to night vision.
- Set resolution to 480p when entering night vision mode.

References

[1] Girolami, Michele, et al. "The Integration of ZigBee with the GiraffPlus Robotic Framework." Evolving Ambient Intelligence. Springer International Publishing, 2013. 86-101.
[2] http://playframework.com/
[3] http://www.arduino.cc/
[4] https://developer.getpebble.com/2/guides/
[5] https://pushover.net/
[6] Koshmak, Gregory, Maria Linden, and Amy Loutfi. "Dynamic Bayesian Networks for Context-Aware Fall Risk Assessment." Sensors 14.5 (2014): 9330-9348.
[7] Palumbo, Filippo, et al. "Sensor network infrastructure for a home care monitoring system." Sensors 14.3 (2014): 3833-3860.
[8] http://www.apache.org/licenses/LICENSE-2.0.html
[9] https://play.google.com/store
[10] http://www.mongodb.org/
[11] http://rtyley.github.io/spongycastle/
[12] Barsocchi, Paolo, et al. "Evaluating Ambient Assisted Living Solutions: The Localization Competition." Pervasive Computing, IEEE 12.4 (2013): 72-79.
[13] Palumbo, Filippo, et al. "Multisensor data fusion for activity recognition based on reservoir computing." Evaluating AAL Systems Through Competitive Benchmarking. Springer Berlin Heidelberg, 2013. 24-35.
[14] http://ziyang.eecs.umich.edu/projects/powertutor/