# Teseo: a multi-agent tracking application in wireless sensor networks

Filippo Zanella and Angelo Cenedese

***Abstract*—**In this work the design and implementation of an application to track multiple agents in a indoor Wireless Sensor Actor Network (WSAN) is proposed. We developed a tracking algorithm that falls into the category of the radio frequency localization/tracking methods, that exploit the strength of the wireless communications among fixed and mobile agents to establish the position of the mobile ones. The algorithm resorts to an Extended Kalman Filter to process the agents measurements and reach a desired level of tracking performance. The tracking application, namely Teseo, is composed by a low-level NesC management software for the agents side and a Java graphical interface provided to users connected to mobile agents. A detailed description of the operations performed by Teseo is given, accompanied both by simulations to validate the tracking algorithm and experiments on a real testbed to test Teseo.

***Keywords*—**Wireless sensor network, tracking, localization, Kalman filter, embedded systems, TinyOS, NesC

## I. Introduction

In recent years, the employment of Wireless Sensor Actor Networks (WSANs) to gather data from the environment have been increasingly envisaged for building management systems and environment control [1], [2], thanks to their versatility of use, easiness of deployment, pervasiveness of data, adaptability to system/environment variations [3], [4], [5]. Examples in this sense are given by Heating and Ventilation Air Conditioning (HVAC) systems [6] employing more and more advanced control techniques that would benefit from a detailed mapping of the internal building parameters; by event detection and surveillance systems, where the heterogeneity of agent devices and the computational grid created by the network itself allow the definition of data fusion policies [7], [8]; by localization and tracking systems where the wireless devices can exploit the received power signal during broadcast/peer-to-peer communication to perform position estimation [9].

The growing interested for the WSANs has been supported by the diffusion of small and cheap devices, capable of radio frequency (RF) communication, computation, and memory, although of limited resources. An example in this sense is the Tmote Sky [10], an ultra low power IEEE 802.15.4 compliant wireless device, which has become a reference in the academia for the early development of algorithms and applications for WSANs. These devices are based on the TinyOS operative system [11] and are programmed in NesC [12], a C-derived language specifically developed for embedded systems.

F. Zanella and A. Cenedese are with the Information Engineering Department, University of Padova, via Gradenigo 6/B, 35131, Padova, ITALY (phone: 0039-049-8277600; fax: 0039-049-8277699; e-mail: filippo.zanella@dei.unipd.it, angelo.cenedese@unipd.it).

### *A. Contribution*

The work presented in this paper belongs to framework of the RF-based localization and tracking, and in particular to the multi-agent tracking problem, where a set of mobile devices (i.e. mobile nodes) are moving within a network of fixed (and known) position similar devices (i.e. fixed nodes), with which they communicate through a RF channel exchanging information on the surrounding.

In this paper we introduce an easy to implement and fast responsive Extended Kalman Filter (EKF) approach for the RF-based localization and tracking, and we describe the implementation stages of Teseo application we developed, which is a combination of NesC and Java software. We show how the implementation in this framework is particularly challenging since the tracking procedure requires correct communication, scheduling, and synchronization among the devices to work properly and attain the expected performance. Moreover, the limited resources available to the embedded devices calls for efficient coding solutions, both in terms of memory and computational power. The code is available freely as open-source on Sourceforge [13], distributed under the GNU General Public License, Version 3, 29th June 2007, whom copyrights are owned by the Free Software Foundation.

### *B. State of the art*

In the framework of distributed systems composed of not-expensive embedded devices, one immediate advantage of RF-based tracking with respect to other methodologies is that the former does not need additional hardware components such as ultrasound, infrared, or light modules, to generate the localization signal that is then measured to compute the angle of arrival, the time of arrival, or time difference of arrival [14].

Differently, the RF-based method parasitically exploits the communication flow that is anyway ongoing among the nodes, and the measurement techniques is relying on the Radio Signal Strength (RSS) either basically inverting the relation between the distance and the received power (radio-channel model), or matching the received power with a pre-compiled map of the environment linking power values to positions. Common references for the former range-based methods and the latter range-free methods are respectively [15] and [16].

In this context, it appears how the accuracy in the localization/tracking strongly depends on the quality of the specific embedded hardware devices and how the algorithmic solutions aim at providing software correction procedure to improve the basic performance of the system.

In particular, a solution is sought that, whereas guaranteeing a certain level of tracking accuracy, is easy to implement, does not require high resources to the embedded device, is robust to failures, and quick enough to converge for real-time use.

### C. Paper organization

Sec. 2 introduces the channel model adopted by the tracking algorithm, while in Sec. 3 we describe our proposed algorithm for determining mobile nodes positions through an EKF [17] approach. Sec. 4 briefly explains the interactions between the agents and the client, Sec. 5 and Sec. 6 are dedicated to the explanation of the design of Teseo both for the NesC and Java coding. Sec. 7 contains simulations of the core algorithm based on an exemplary WSAN configuration. Sec. 8 concludes.

In general, we will use bold fonts to indicate vectorial quantities, plain italic fonts to indicate scalar ones, capital vertical fonts to indicate matrices.

## II. CHANNEL MODELING

The performances of tracking algorithms are influenced by the effects of noises and disturbances introduced into the communication channel, so it is necessary to identify these contributions as accurately as possible [18]. The measurements of received power exchanged by agents in a Wireless Sensor Actor Network (WSAN) are affected by objects in the environment (such as walls or furniture) that cause attenuation, reflection, diffraction and diffusion effects. Moreover, errors that vary over time are caused by generic noises and interferences. Based on these considerations, we present a general channel model which takes into account different kind of disturbances. Then we focus on a reduced channel model, subject to particular assumptions, that we employ to design the multi-agent tracking algorithm.

### A. General model

As we previously stated, to model the channel in an indoor environment it is necessary to consider different factors: the free-space path loss, that expresses the power loss due to dissipation of energy in the channel, the fading phenomena, like shadowing and multi-path, that express the variability of the channel.

A WSAN is usually treated as a graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$, where the set $\mathcal{N}$ of the nodes (i.e. agents) communicate along the edges (i.e. communication links) specified by the set $\mathcal{E}$. Given a node $i$, the set $\mathcal{V}(i) := \{j \,|\, (i,j) \in \mathcal{E}, i \neq j\}$ collects its neighbors.

In our context the WSAN is primarily composed of a set $\mathcal{F}$ of $F$ nodes in fixed positions, that do not know a priori their neighbors $\mathcal{V}(i)$, $i = 1, \ldots, F$, but instead they know their positions $\boldsymbol{z}_i := (\overline{x}_i, \overline{y}_i)$, in the 2-dimensional space.

A well agreed channel model is the log-distance path loss model [19], where the received power is linked to the transmission power through a log-normal model of path loss, and other contribution terms are added to take into account of the other disturbing effects. The model that describes the wireless channel between two nodes, in terms of received power $P_{ij}$, is the following [20]:

$$P_{ij} := P_j^{tx} + r_j + f_{pl}(d_{ij}) + f_{sf}(\boldsymbol{z}_i, \boldsymbol{z}_j) + f_a(\boldsymbol{z}_i, \boldsymbol{z}_j) + v_{ff}(t) + o_i, \quad (1)$$

where $i$ and $j$ are the receiver and transmitter node, at a distance $d_{ij} := \|\boldsymbol{z}_i - \boldsymbol{z}_j\|$ ($\|\cdot\|$ being the classical Euclidean norm). Moreover, $P_j^{tx}$ is the transmitted power, $r_j$ is the transmission offset between the nominal and the effectively transmitted power (which is usually reported in the datasheet of the devices); $f_{pl}(\cdot)$ represents the path loss; $f_a(\cdot)$ represents the channel asymmetry factor; $f_{sf}(\cdot)$ models the slow fading components while the $v_{ff}(\cdot)$ represents the fast ones, and $o_i(\cdot)$ represents the measured received strength offset of the receiving node.

### A. Simplified model

The parameters of (1) depend on the environment where the WSAN is deployed and the specific hardware of the wireless devices. In general, to perform a channel parameter identification, the model of (1) is simplified assuming that the transmission power of the sensors is set at the maximum level (i.e. $P_j^{tx} = 0$ dBm, $\forall j \in \mathcal{N}$) so that the transmitter offset is almost zero, $r_j \cong 0$ dBm. Furthermore, we consider that $o_i = 0$, $\forall i \in \mathcal{N}$, since the offsets can be easily compensated exploiting a distributed strategy[1] [20].

Lastly, the fast fading effect $v_{ff}(t)$ is removed, by averaging the received power over a set of $0 < C_{ij} \leq C$ consecutive measures: $\overline{P}_{ij} := \sum_{k=1}^{C_{ij}} P_{ij}^k$.

It follows that the average received power $\overline{P}_{ij}$ becomes:

$$\overline{P}_{ij} = \beta - 10\gamma \log_{10}(d_{ij}) + f_{sf}(\boldsymbol{z}_i, \boldsymbol{z}_j) + f_a(\boldsymbol{z}_i, \boldsymbol{z}_j). \quad (2)$$

Since the components of slow fading and channel asymmetry are independent Gaussian random variables of variance $\sigma_{sf}^2$ and $\sigma_a^2$ respectively, they can be combined into one zero-mean random variable $q_{ij}$ with variance equal to $\sigma^2 = \sigma_a^2 + \sigma_{sf}^2$:

$$\overline{P}_{ij} = \beta - 10\gamma \log_{10}(d_{ij}) + q_{ij}. \quad (3)$$

[1] Experimental evidence indicates that agent offsets $o_i$ are not negligible and can be substantially large for some nodes (up to 6 dBm). The effect of this offset is to bias the estimate of the distance between two nodes, which is particularly harmful in tracking applications.

From (3) it is clear that $\beta$ and $\gamma$ are the only parameters that determine the model of the communication channel. Being the components of slow fading and channel asymmetry independent Gaussian random variables, we can use a (distributed) least-squares estimator to estimate those parameters, as it has been addressed in [19].

## III. TRACKING ALGORITHM

In this section we describe our proposed algorithm for determining mobile nodes positions.

Suppose that in the WSAN a set $\mathcal{M}$ of $M$ mobile nodes can freely move. Thus the WSAN is overall constituted by $F + M$ agents, in the set $\mathcal{N} = \mathcal{F} \oplus \mathcal{M}$.

The proposed algorithm, that allows to estimate the 2-dimensional position of a mobile node is based on the assumptions that, at each time step $k$, $k \in \mathbb{Z}$ each mobile node $m$, $m \in \mathcal{M}$, knows:

- the coordinates $\boldsymbol{z}_n$ of each fixed node $n \in \mathcal{F}$;
- the average power $\overline{P}_{mn}$ received from each $n \in \mathcal{V}_k(m)$ fixed node over a period of time$[(k-1)\ \ k]$, where $\mathcal{V}_k(m)$ is the set of the $F_k$ neighbors of node $m$ in the period $[(k-1)\ \ k]$ (notice that $F_k \leq \dim(\mathcal{F})$ changes at each time step $k$);
- the channel parameters $\beta$ and $\gamma$;

Aim of the algorithm is the disjoint estimation of the coordinates $\boldsymbol{\xi}_m(k) := (x_m(k), y_m(k)) \in \mathbb{R}^2$ of the mobile nodes, at each time step $k$.

### A. State-space model

Define the quantities

$$\mathrm{Z}(k) := \begin{bmatrix} \boldsymbol{z}_{n_1} \\ \vdots \\ \boldsymbol{z}_{n_{F_k}} \end{bmatrix} \in \mathbb{R}^{F_k \times 2}$$

$$h_{n_i}\left(\boldsymbol{\xi}_m(k), \boldsymbol{z}_{n_i}\right) := \beta - 10\gamma \log_{10}\left(\|\boldsymbol{\xi}_m(k) - \boldsymbol{z}_{n_i}\|\right) \in \mathbb{R}$$

$$\mathbf{h}\left(\boldsymbol{\xi}_m(k), \mathrm{Z}(k)\right) := \begin{bmatrix} h_{n_1}\left(\boldsymbol{\xi}_m(k), \boldsymbol{z}_{n_1}\right) \\ \vdots \\ h_{n_{F_k}}\left(\boldsymbol{\xi}_m(k), \boldsymbol{z}_{n_{F_k}}\right) \end{bmatrix} \in \mathbb{R}^{F_k}$$

and

$$\boldsymbol{\psi}(k) := \begin{bmatrix} \overline{P}_{mn_1}(k) \\ \vdots \\ \overline{P}_{mn_{F_k}}(k) \end{bmatrix} \in \mathbb{R}^{F_k}$$

For each mobile node $m$ we have the state model:

$$\boldsymbol{\xi}_m(k+1) = \mathrm{A}\,\boldsymbol{\xi}_m(k) + \mathbf{w}(k) = \boldsymbol{\xi}_m(k) + \mathbf{w}(k) \qquad (4)$$

and the measurements model:

$$\boldsymbol{\psi}(k) = \mathbf{h}\left(\boldsymbol{\xi}_m(k), \mathrm{Z}(k)\right) + \mathbf{v}(k) \qquad (5)$$

where $\mathrm{Z}(k)$ is the matrix of known positions $\boldsymbol{z}_{n_i}$, with $i = 1, \ldots, F_k$ of the fixed nodes and $\boldsymbol{\xi}_m(k)$ is the state of the system, i.e. the 2-dimensional position of each mobile node $m \in \mathcal{M}$; $\boldsymbol{\psi}(k)$ is the output of the system, made of $F_k$ powers stored by the mobile node and available at the time $k$. The process noise $\mathbf{w}(k)$ and the measurement noise $\mathbf{v}(k)$ are uncorrelated, white, with zero mean and variance $\mathrm{W} \in \mathbb{R}^{2\times 2}$ and $\mathrm{V}(k) \in \mathbb{R}^{F_k \times F_k}$, respectively.

As we can see, the state transition model is linear and the matrix A is the identity matrices, denoting a typical behavior of a simple random walk. Thus the mobile node is represented as a point mass moving on the 2-dimensional plane, surrounded by a cloud of Gaussian uncertainty.

The model of the measures is rather constituted by the channel model (3), which is non-linear. Notice how the measurement model is time variant, i.e. its dimension varies at each time step $k$ according to the number $F_k$ of the collected power measurements. Specifically, at each time step $k$ a mobile node $m$ collects $F_k$ averaged measurements $\overline{P}_{mn_i}(k)$ from its dynamic neighbors $n_i \in \mathcal{V}_k(m)$, $i = 1, \ldots, F_k$.

### B. Structure of the algorithm

Assume without loss of generality that $\dim(\mathcal{M}) = 1$. We define $\boldsymbol{\xi}(k) := \boldsymbol{\xi}_m(k)$ to indicate the position of the only mobile node $m \in \mathcal{M}$. The idea behind the algorithm is to operate two different types of filtering depending on the number $F_k$. If $F_k < 3$ the mobile node updates its state following an open-loop approach, otherwise it uses an EKF technique.

The choice of two approaches derives from the fact that we want to provide the EKF a minimum number of measures to update the estimate $\widehat{\boldsymbol{\xi}}(k|k)$. That minimum has been arbitrarily set equal to 3, recalling somewhat the constraint that appears in the algorithms based on trilateration/triangulation methods. If the measures available in the various sampling instants are less than 3, the algorithm expects to leave the filter in an open loop. The mobile node continues to regard as an estimate of the current position the last estimated position based on measurements received, but increasing step by step the variance of the filtering error. This approach forces the filter to consider the mobile node still in the same position both if there is packet loss (or the mobile node is simply in a dead zone) and if the acquired measurements are somehow corrupted.

Now let's see in detail the two types of filtering presented. Every period $[(k-1)\ \ k]$ the mobile node $m$ identifies the set $\mathcal{V}_k(m)$, i.e. the $F_k$ neighboring nodes, based on the measurements that it has collected in that time interval.

If $F_k \geq 3$ the function $\mathbf{h}(\cdot)$ is linearized near the point $\widehat{\boldsymbol{\xi}}(k|k-1)$, which is the best estimation of the mobile node state at the instant $k$. Then the Jacobian:

$$\mathrm{H}(k) = \left[\frac{d\mathbf{h}(\boldsymbol{\xi},\cdot)}{d\boldsymbol{\xi}}\right]\Bigg|_{\boldsymbol{\xi}=\widehat{\boldsymbol{\xi}}} \in \mathbb{R}^{F_k\times 2} \tag{6}$$

is computed, which yields

$$\mathrm{H}(k) = -\frac{10\gamma\log_{10}\mathrm{e}}{\left\|\mathbb{1}\widehat{\boldsymbol{\xi}}(k|k-1)-\mathrm{Z}(k)\right\|^2}\left(\mathbb{1}\widehat{\boldsymbol{\xi}}(k|k-1)-\mathrm{Z}(k)\right)$$

Then, the minimum variance linear estimator $\widehat{\boldsymbol{\xi}}(k|k)$ of the state $\boldsymbol{\xi}(k)$, based on the observations $\boldsymbol{\psi}(k)$, is computed through the recursive algorithm:

$$\begin{aligned}
&\Lambda(k)=\mathrm{H}(k)\,\mathrm{Q}(k|k-1)\,\mathrm{H}(k)^T+\mathrm{V}(k)\\
&\mathrm{L}(k)=\mathrm{Q}(k|k-1)\,\mathrm{H}(k)^T\,\Lambda(t)^{-1}\\
&\mathrm{Q}(k|k)=\mathrm{Q}(k|k-1)-\mathrm{Q}(k|k-1)\,\mathrm{H}(k)^T\,\Lambda^{-1}\,\mathrm{H}(k)\,\mathrm{Q}(k|k-1)
\end{aligned}$$

where the minimum variance linear predictor $\widehat{\boldsymbol{\xi}}(k+1|k)$ is given by

$$\begin{aligned}
&\widehat{\boldsymbol{\xi}}(k+1|k)=\mathrm{A}\widehat{\boldsymbol{\xi}}(k|k)=\widehat{\boldsymbol{\xi}}(k|k)\\
&\mathrm{Q}(k+1|k)=\mathrm{A}\,\mathrm{Q}(k|k-1)\,\mathrm{A}^T+\mathrm{W}=\mathrm{Q}(k|k)+\mathrm{W}
\end{aligned}$$

with $\Lambda(k)$ variance of the innovation process $\mathbf{e}(k)=\boldsymbol{\psi}(k)-\mathrm{H}(k)\widehat{\boldsymbol{\xi}}(k|k-1)$ and $L(k)$ gain of the filter.

If $F_k < 3$ we have:

$$\begin{aligned}
&\widehat{\boldsymbol{\xi}}(k|k)=\widehat{\boldsymbol{\xi}}(k|k-1) \qquad \widehat{\boldsymbol{\xi}}(k+1|k)=\widehat{\boldsymbol{\xi}}(k|k)\\
&\mathrm{Q}(k|k)=\mathrm{Q}(k|k-1) \qquad \mathrm{Q}(k+1|k)=\mathrm{Q}(k|k)+\mathrm{W}
\end{aligned}$$

that jointly become:

$$\begin{aligned}
&\widehat{\xi}(k+1|k)=\widehat{\xi}(k|k-1)\\
&\mathrm{Q}(k+1|k)=\mathrm{Q}(k|k-1)+\mathrm{W}
\end{aligned}$$

outlining clearly the effect of the stationary solution.
The scheme of the algorithm is summarized by Alg. 1 in Fig. 1.

The use of the EKF approach lies on the fact that it is easy to implement and it does not require significant computational resources, thanks to the structure of the filter itself and to the size of the system. The proposed variant EKF is intrinsically time-varying and it does not admit regime solutions, even if the system is stable, but nothing can be said regard observability (of the linearized system, because $F_k$ is variable). Therefore, as it is well-known, there is no guarantee that the EKF converge.

The initial conditions of the algorithm are defined as $\widehat{\boldsymbol{\xi}}(0|-1)=\boldsymbol{\mu}_0$ $\mathrm{Q}(0|-1)=\mathrm{Q}_0$, with $\boldsymbol{\mu}_0=\mathbb{E}[\boldsymbol{\xi}(0)]$ and $\mathrm{Q}_0=\mathrm{var}\{\boldsymbol{\xi}(0)\}$. Since these quantities are not known in advance, specific estimation techniques can be used to get a guess. Trilateration, bounding box or least-square methods are some of the simplest and most popular for estimating the initial position [21].

**Algorithm 1** Generic mobile node tracking

1: $\widehat{\boldsymbol{\xi}}(k|k)\in\mathbb{R}^2$; $\mathrm{Q}(k|k)\in\mathbb{R}^{2\times 2}$ and $k=0,2,\ldots$

2: $\sigma_w\in\mathbb{R}$, noise model variance
3: $\sigma_v\in\mathbb{R}$, noise measure variance

4: set: $\widehat{\boldsymbol{\xi}}(0|-1)=\boldsymbol{\mu}_0$; $\mathrm{Q}(0|-1)=\mathrm{Q}_0$; $\mathrm{W}=\sigma_w^2\mathrm{I}_2$

5: set up of the measurements data set, collecting $F_k$ power transmission from neighboring nodes.

6: **for** $k=0,2,\ldots$ **do**
7: $\mathrm{V}(k)=\sigma_v^2 I_{F_k}$
8: **if** $F_k\geq 3$ **then**
9: $\Lambda(k)=\mathrm{H}(k)\,\mathrm{Q}(k|k-1)\,\mathrm{H}(k)^T+\mathrm{V}(k)$
10: $\mathrm{L}(k)=\mathrm{Q}(k|k-1)\,\mathrm{H}(k)^T\,\Lambda(t)^{-1}$
11: $\widehat{\boldsymbol{\xi}}(k|k)=\widehat{\boldsymbol{\xi}}(k|k-1)+\mathrm{L}(k)\left[\boldsymbol{\psi}(k)-\mathbf{h}\left(\widehat{\boldsymbol{\xi}}(k|k-1)\right)\right]$
12: $\mathrm{Q}(k|k)=\mathrm{Q}(k|k-1)-\mathrm{Q}(k|k-1)\,\mathrm{H}(k)^T\,\Lambda^{-1}\,\mathrm{H}(k)\,\mathrm{Q}(k|k-1)$
13: **else**
14: $\widehat{\boldsymbol{\xi}}(k|k)=\widehat{\boldsymbol{\xi}}(k|k-1)$
15: $\mathrm{Q}(k|k)=\mathrm{Q}(k|k-1)$
16: $\widehat{\boldsymbol{\xi}}(k+1|k)=\widehat{\boldsymbol{\xi}}(k|k)$
17: $\mathrm{Q}(k+1|k)=\mathrm{Q}(k|k)+\mathrm{W}$

Fig. 1 scheme of the tracking algorithm for a generic mobile agent

The use of the EKF requires knowledge of the standard deviation $\sigma_w$ of model noise $\mathbf{w}(k)$ and standard deviation $\sigma_v$ of measurement noise $\mathbf{v}(k)$. Regarding $\sigma_w$ we opted for an empirical calibration. Assuming that the mobile node is anchored to a human user, its variance, at each $[(k-1)\quad k]$, can be set equal to that associated to a typical human motion, and therefore to define the diagonal elements of $\mathrm{W}$. If we considered the fastest man in the world, with a sampling time of $60$ ms between two consecutive estimations, the variance model would correspond to $0.3844$ m$^2$, which can be thought as an upper bound to the variance. $\sigma_v$ is usually available from the specific of sensing device with whom measurements are performed. Since, in this case, the measuring instrument is the communication channel, all the variances of the fading effects and asymmetry of the channel should be accurately evaluated. In Sec. 4 a practical example for a specific device is given.

## IV. Software Design

A set of indexed mobile nodes $\mathcal{M}=\{m_1,\ldots,m_M\}\subseteq\mathbb{N}$ moves within a network of indexed fixed nodes $\mathcal{F}=\{f_1,\ldots,f_F\}\subseteq\mathbb{N}$, each node running a TinyOS

module and communicating via wireless, assuming the parameters of the radio channel as known [20].

Also, each mobile node is connected to a client (laptop) through a USB connection, with the client performing the multi-agent tracking (MAT) computation envisaged by the algorithm in Sec. 3 and implementing Java classes for the Graphical User Interface (GUI).

When one (or more) mobile node $m_i$ starts the tracking process:

1. every $T_s$ ms $m_i$ alerts its client $C_{m_i}$ to be ready, sending via USB $PCM_{max}$ pings every $T_p$ ms; afterwards, $m_i$ sends via wireless $PNM_{max}$ pings every $T_n$ ms;
2. as $C_{m_i}$ receives a ping from $m_i$, it enables a timer that starts the MAT procedure every $T_c$ ms;
3. the set of fixed nodes $\{f_i\}$ that gets in touch with $m_i$ starts to broadcast $DM_{max}$ messages every $T_t$ ms, for a period not exceeding $T_s$ ms;
4. $m_i$ stores one by one the messages received from the $\{f_i\}$, filtering them according to a predefined Receive Signal Strength (RSS) threshold ($RSS_{bound}$), and forwards these messages to $C_{m_i}$;
5. $C_{m_i}$ stores the messages and every $T_c$ ms estimates the position of $m_i$, showing it in a GUI.

Fig. 2 outlines the schema of MAT scheduling, for a complete cycle of the algorithm of $T_s = TIMER\_STEP$ ms. It compares with the same time scale the operating modes of the fixed nodes, the mobile node and the client. Scheme of Fig. 2, although complete, is simplified, as it does not highlight the randomness linked to the execution of some events. However, it is significant for understanding the temporal evolution of the processes that constitute the main algorithm.

The whole software can be divided into two main blocks, according to the programming language: NesC for the nodes and Java for the client. Since in the considered context the peer-to-peer behavior among nodes appears of major interest, it will be dealt more in detail in the remainder of the paper.

## V. Implementation: NesC for nodes

Four message types are defined to exchange information among different devices Fig. 3:

- *mote_ctrl_msg*, to start/stop the MAT process. A stop signal interrupts any communication in progress; vice versa, a start forces mobile nodes to begin a new cycle of the algorithm. This message is sent via USB from $C_{m_i}$ to $m_i$;
- *ping_client_msg*, to ping the clients. It is used by $m_i$ to inform $C_{m_i}$ that a MAT is ready to start and to send configuration settings. This message is sent via USB from $m_i$ to $C_{m_i}$; *ping_node_msg*, to ping fixed nodes. It is used by $m_i$ to ping the $\{f_i\}$ in the communication ranges. This message is broadcast by $m_i$ via radio;
- *data_msg*, to measure RSS values. When $m_i$ receives this message, it computes RSS and sends the information to $C_{m_i}$, enabling the position estimate. This message is broadcast via radio by $\{f_i\}$ to $m_i$ and via USB by $m_i$ to $C_{m_i}$.

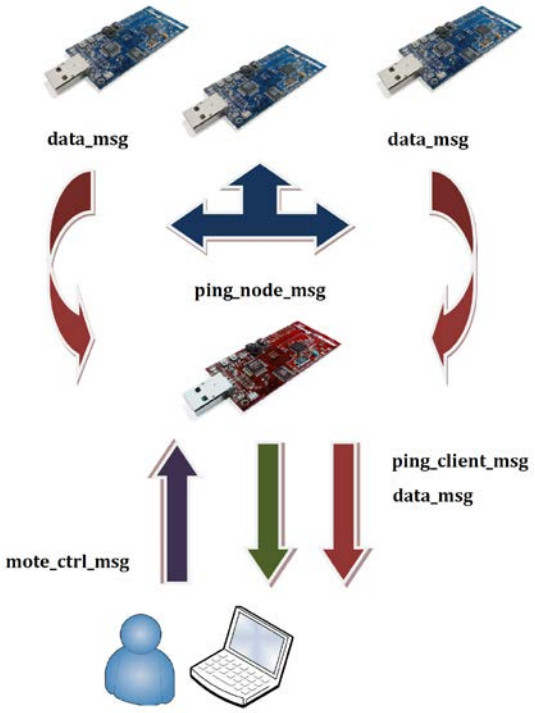


Fig. 3 Messages exchange between devices. Red arrows indicate *data_msg*, purple arrow *mote_ctrl_msg*, green arrow *ping_client_msg* and blue arrows *ping_node_msg*.

To avoid potential overlaps among tasks, commands or events related to various operation states of the nodes, nodes are treated as finite state machines, implying that the operations of different node states cannot interfere with each other. The feasible states of fixed nodes $\{f_i\}$ are:

- *IDLE*: inactivity;
- *TRANSMISSION*: broadcasting *data_msg*;

while mobile node $m_i$ is characterized by the states:

- *SEND_CLIENT*: sending *ping_client_msg*;
- *SEND_NODE*: sending *ping_node_msg*;
- *AUDIT_NODE*: auditing *data_msg*;
- *DO_NOTHING*: inactivity.

In addition, $m_i$ is enabled/disabled by $C_{m_i}$ through the following commands:

- *START_MN*: starts mobile node;
- *STOP_MN*: stops mobile node.

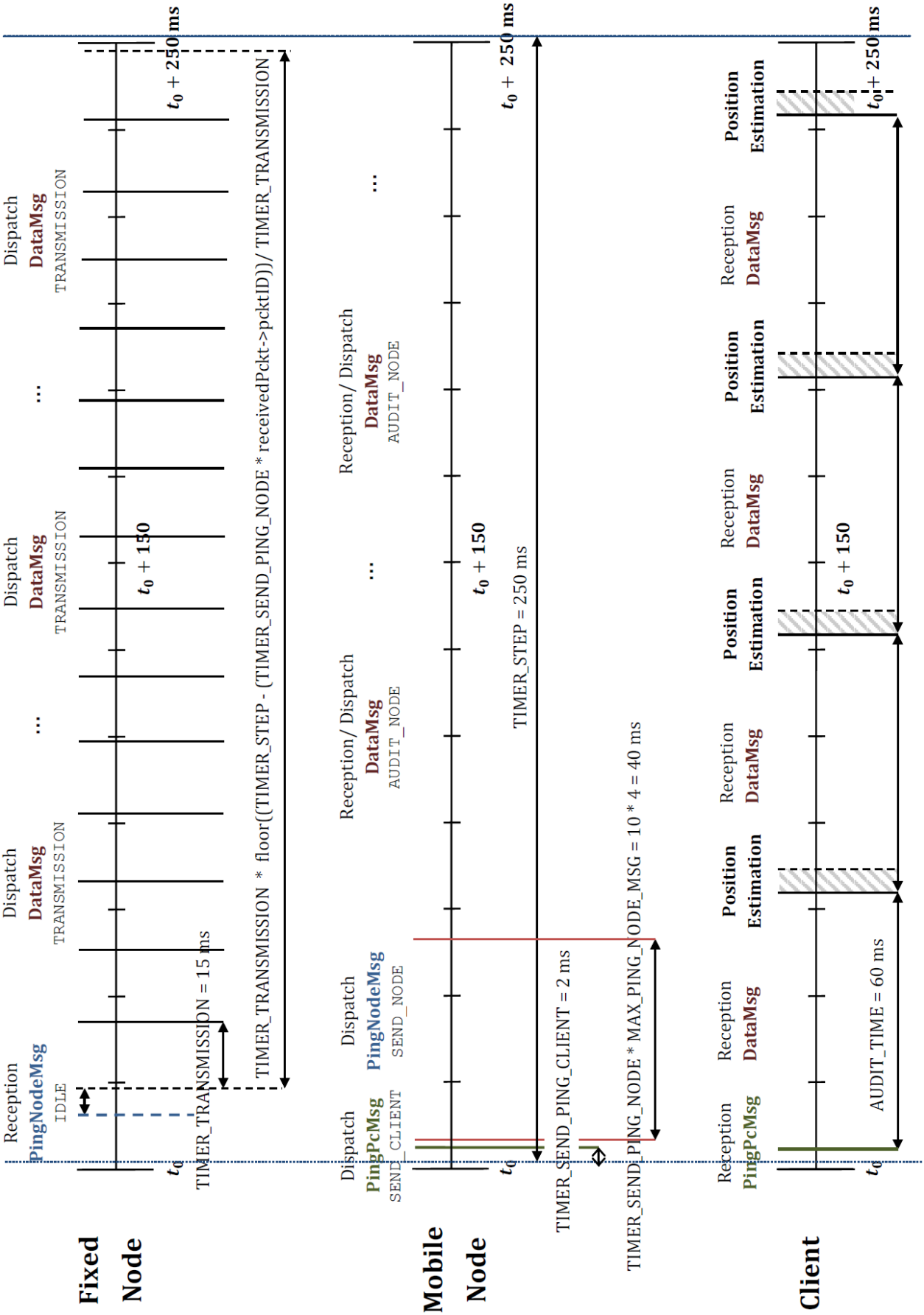


Figure 2 Scheduling of tasks, timers, and communication events of node and client devices during MAT

### A. Mobile node activity

To understand through an example the function covered by each of the routine of module *MobileNodeP*, involved in the MAT algorithm, we simulate a normal operation of the mobile node during the tracking procedure. In the description of the source code will not be mentioned TinyOS modules of *MobileNodeP*: they are an integral part of the configuration file *MobileNodeC* whose purpose is to delineate both the programmer and the compiler how the various components are interconnected.

#### B.1. Boot

When a mobile node $m_i$ is turned on, the boot sequence commences. In the function *booted()* of interface **Boot** peripherals and environment are initialized, moving $m_i$ in the states *DO_NOTHING* and *WAIT_CMD*: $m_i$ waits to receive a *START_MN* command by client $C_{m_i}$.

It is also enabled the user button of the mobile node to allow user to start, *START_MN*, or stop, *STOP_MN*, the mobile node regardless of the client.

The transmission frequency is set to *CHANNEL_RADIO* by command *setChannel(uint8_t)* of **CC2420Config** interface. If the event *syncDone(error_t)* signals that the routine is terminated correctly then radio and serial communication are turned on.

If all operations are carried out properly all the LEDs are switched on, otherwise it is sufficient that any one LED is off to indicate that there is a problem in the init. Notified events *startDone(error_t)* of CC2420 and RS232 peripherals, a call to setPower(message_t*, uint8_t) of **CC2420Packet** sets to *POWER_RADIO* the transmission power of *ping_node_msg*.

After this operations, mobile node waits for a command from the client side.

#### B.2. Clock Step

When $m_i$ receives a *START_MN* from $C_{m_i}$, it starts the timer *ClockStep* that every $T_s =$ *TIMER_STEP* ms launches the *fired()* event. With this instance, the MAT algorithm begins: $m_i$ moves to the *SEND_CLIENT* state, all packets counters are reset, and timer *ClockSendPingClient* starts.

#### B.3. Clock Send Ping Client

When $T_p =$ *TIMER_SEND_PING_CLIENT* ms elapse, $C_{m_i}$ is repeatedly informed of the start of the MAT process, for a number of times equals to $PCM_{max} =$ *MAX_PING_CLIENT_MSG*. This activity is performed by posting task *sendPingClientMsg()*, which forwards messages *ping_client_msg* to the serial port. Then $m_i$ moves to the *SEND_NODE* state, stops the timers related to *ping_client_msg* sending, and starts the timer *ClockSendPingNode*.

#### B.4. Clock Send Ping Node

When $T_n =$ *TIMER_SEND_PING_NODE* ms are elapsed, task *sendPingNodeMsg()*, periodically posted by the timer, broadcasts $PNM_{max} =$ *MAX_PING_NODE_MSG* messages of type *ping_node_msg*, specifying the identification number (ID) *TOS_NODE_ID* of the node $m_i$ and the settings of the selected transmission channel. When $m_i$ stops to ping fixed nodes $\{f_i\}$ in range, it moves to the *AUDIT_NODE* states and stops the timer *ClockSendPingClient*. Then it waits to receive *data_msg* messages.

#### B.5. Receive data_msg

The fixed nodes $\{f_i\}$ that receive at least one *ping_node_msg* respond to the mobile node $m_i$ sending their *data_msg* messages. From these messages $m_i$ extracts the values of RSSI, shifted by a *RSSI_OFFSET* offset, using the command *getRssi(int8_t)* of interface **CC2420Packet**. Messages with RSS greater than the threshold *RSS_BOUND* are stored in a FIFO queue, **Queue<data_msg>**, of size *QUEUE_DATA_SIZE*.

Then, $m_i$ invokes task s*endDataMsg()*, which forwards to the serial port all the *data_msg* messages contained in the queue; this is done only if the queue has not already been emptied in a previous sending. $m_i$ remains in the *AUDIT_NODE* state until timer *ClockStep* fires again, hereupon the mobile node returns to the initial conditions, ready to begin a new cycle. Anytime, the user retains the ability of stopping the algorithm execution with the command *STOP_MN*. In this case all timers are stopped and $m_i$ enters the *IDLE* state.

### B. Fixed node activity

Similarly to the previous subsection, to describe the implementation of module *FixedNodeP*, we simulate the normal operation of the routines involved in the MAT algorithm.

#### B.1. Boot

When one fixed node $f_i$ turns on, TinyOS starts the boot sequence. In the function *booted()* peripherals and environment are initialized, moving $f_i$ to the *IDLE* state, meaning that the fixed node $f_i$ waits to receive a *ping_node_msg* message from a mobile node$m_i$, via radio communication.

The transmission frequency is set to *CHANNEL_RADIO* and if the event *syncDone(error_t)* signals that the synchronization has been completed correctly, the radio and serial communication are turned on.

Notified event *startDone(error_t)*, a call of *setPower(message_t*, uint8_t)* sets to *POWER_RADIO* the transmission power of *data_msg* messages. After this operation the fixed node is ready to receive messages from the network.

*B.2. Receive ping_node_msg*

When $f_i$ receives a first *ping_node_msg* from a mobile node$m_i$, identified by a unique $ID[k]$, $k \in [1\ PNM_{max}]$, it starts the timer *TimeToSend* that every $T_l =$ *TIMER_TRANSMISSION* ms launches its event *fired()*. In this stage, before moving to the *TRANSMISSION* state, the node $f_i$ computes the maximum number of *data_msg* to be sent to the mobile node$m_i$, that is given by:

$$DM_{max} := \left\lfloor \frac{T_s - T_n\, ID[k]}{T_t} \right\rfloor,$$

where $T_s$ and $T_n$ are the times previously defined. This action is carried out in order to reduce network traffic. Indeed, in doing so, the fixed node $f_i$ stops the transmission of *data_msg* messages before the mobile node in range $m_i$ enters in the next step of the algorithm. The $DM_{max}$ number is recalculated every time since it is proportional to the $ID[k]$ of the first *ping_node_msg* received, that may change due to the packet loss phenomena affecting in general the wireless channel, and in particular the tracking applications [5]. This bound in the transmission of the *data_msg* message forces $f_i$ to move to the state IDLE after $T_t\, DM_{max}$ ms, here remaining unless it receives other *ping_node_msg* by some moving $m_i$ present in the environment.

*B.3. Clock Send Data Node*

When $T_l$ ms elapse, the task *sendDataMsg()*, periodically posted by the timer, sends $DM_{max}$ *data_msg* messages in broadcast, specifying the *TOS_NODE_ID* of the fixed node $f_i$ and leaving empty the fields reserved to the RSS values.
As $f_i$ ends to transmit, it returns to the *IDLE* state and the timer *TimeToSend* is stopped; then $f_i$ waits for any other message sent by any mobile node$m_i$ in range.

## VI. Implementation: Java for Client

The software client, named Teseo, has to accomplish the following two tasks:

1. executing the MAT algorithm from the data transmitted by the mobile node, based on the network retrieved information;
2. managing the output flow and the system setup phase by means of a friendly user interface.

To provide the user with an intuitive interface a Java frame, instance of the class **JFrame**, has been designed. The package is made of the classes:

- **Teseo**: main frame of the GUI, entry point of the client. It defines the following nested classes:
  - **MapPanel**: panel that displays the graphical elements present in the environment (e.g. fixed nodes, mobile node, planimetry);
  - **EstimateTimerTask**: task that executes the routines of class **Estimation**;
  - **Estimation**: object that collects all the methods and variables to compute the position estimation of the mobile node;
- **Constants**: interface for shared constants;
- **DataMsg**: just alike *data_msg*;
- **MoteCtrlMsg**: just alike *mote_ctrl_msg*;
- **PingClientMsg**: as *ping_client_msg*;
- **Channel**: object to manage the transmission channel model and the characteristic parameters;
- **Node**: object that defines a node as an entity made up of a set of 2-dimensional coordinates and an ID;
- **Coordinate2D**: generic 2-dimensional coordinates;
- **VariantExtendedKalmanFilter2D**: the EKF implementation for the 2-dimensional tracking case described in Sec. 4.

The frame is depicted in figure Fig. 4, where there can be highlighted four basic elements: The *menu bar*, the *command console*, the *graphical environment* and the *informative panel*.

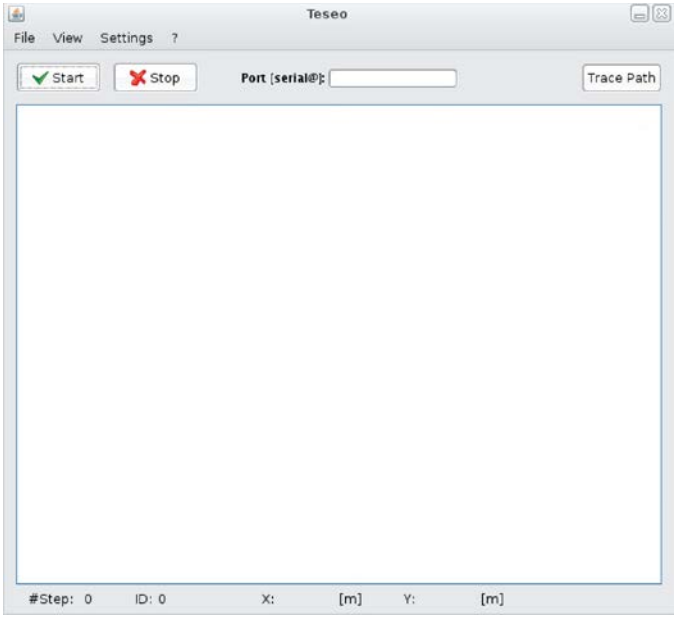

Fig. 4 view of the GUI Teseo

The *menu bar* is made up four items, shown in Fig. 5

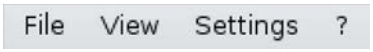

Fig. 5 menu bar of the frame

Clicking on *File* → *Save* a **JFileChooser** appears. It allows to save a text file that is a summary of the mobile node positions estimated by the EKF in the current run. The name of the file is formatted taking into account the current date and hour: *Teseo_<dd_MM_yy-HHmm>.tx*t. From **JMenu** *View* it is possible to show/hide some elements of the frame, like the Verbose System Information (VSI) (linked to the **JCheckBoxMenuItem** *VSI*) and the fixed nodes distributed in the graphical environment (**JCheckBoxMenuItem** *Beacons*). Instead, **JMenu** *Settings* allows to set:

- the parameters $\beta$ and $\gamma$ of the transmission channel, through the **JDialog** of Fig. 6 callable by **JMenuItem** *Channel*. The parameters of the **JDialog**, as they appears in Fig. 6, are initialized in the method *initMyComponents()* of the frame;

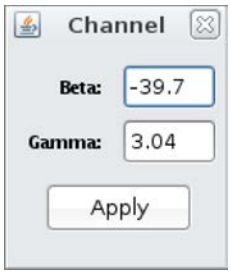


Fig. 6 Dialog window for the configuration of the channel

- times $T_u$ ($\propto$ *K_T_U* ms) and $\tau_u$ ($\propto$ *K_TAU_U* ms), that are respectively the refresh time of the GUI, i.e. the sampling time of the mobile node position visualization, and the delay with whom the trace of the path of the mobile node starts to be plotted (the delay corresponds to the $\tau_u$ ms after the reception of the first **PingClientMsg**). Both values can be chosen by the scrollable bars of the dialog windows associated to the **JMenuItem** *Trace*. Values assigned to $\tau_u$ and $T_u$ of **JDialog** in Fig. 7 cames from the default initialization brought by *initMyComponents()* method.


Fig. 7 dialog window of the trajectory settings $T_u$ and $\tau_u$

The *graphical environment* is a **MapPanel**, extension of the class **JPanel**, that collects a set of methods to show the movement in $\mathbb{R}^2$ of the mobile node in the surrounding environment. It consists of the layout of the building in which are positioned the nodes and of a set of icons useful to point the positions of the fixed nodes and the different positions of the mobile node. In Fig. 8 is given an example of the **MapPanel** during a MAT process of a single mobile node.

The *command console* of Fig. 9 allows to interact with the mobile node, specifying the virtual serial port of the client to which the mobile node is connected. Buttons *Start* and *Stop* are used to start/stop the communication between frame and mobile node.


Fig. 9 consolle di comando del frame

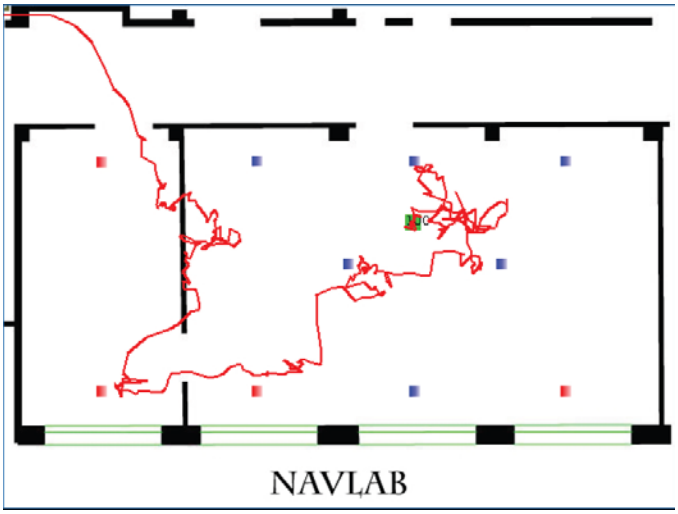


Fig. 8 graphical environment of the frame. The red squares are the active fixed nodes at that time, the blue squares are the inactive fixed nodes, the green square is the mobile node and the red path is the trajectory of the mobile node

In Fig. 10 there are shown the flow charts of the routines *start()* and *stop()*.

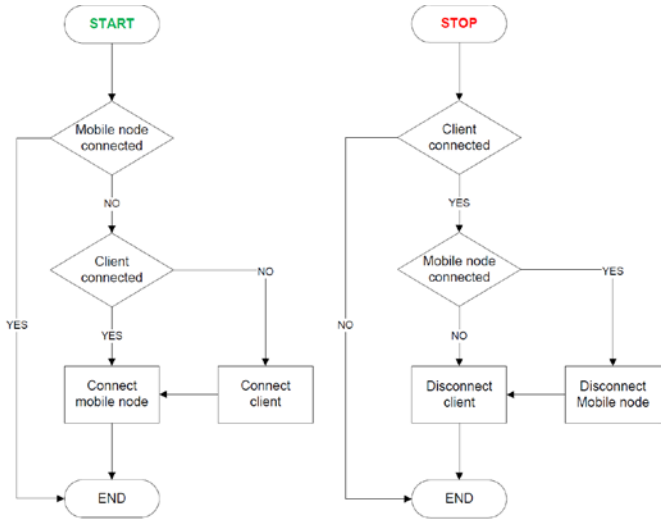


Fig. 10 Flowchart of the start/stop of the mobile node and the client

The *informative panel* displays the numerical value of the 2-dimensional coordinates of the mobile node estimate locally by the running MAT algorithm. It also shows the ID of the mobile node and the number of steps performed by the mobile node that has been notified to the client. Pressing the **JToggleButton** *Trace Path* the tracing option can be enabled/disabled.

The constructor of the frame *Teseo()* is the first method automatically called by the Java virtual machine, therefore it is used to initialize the form. The init is divided into a design side, which instantiates the Swing and AWT palette of the frame via the method *initComponents()*, and a source side, which is related to the method *initMyComponents()*. The source side resets the estimation, allocates and initializes the variables and

objects to it in charge, and completes the instantiation of a set of elements of the frame:

- *VSI update*.
  The *Verbose System Information* is displayed. This is a **JTextArea** within a **JScrollPane**, placed in a **JDialog** external to the frame, which acts as output both to report any anomalies in the use of the client, as the occurrence of some **Exception**, and to tell the user information about the client, such as setting parameters intrinsic to it. The VSI can be hidden/shown through the option *Settings* → *View* of the menu. Fig. 11 gives an example of a VSI information flow;

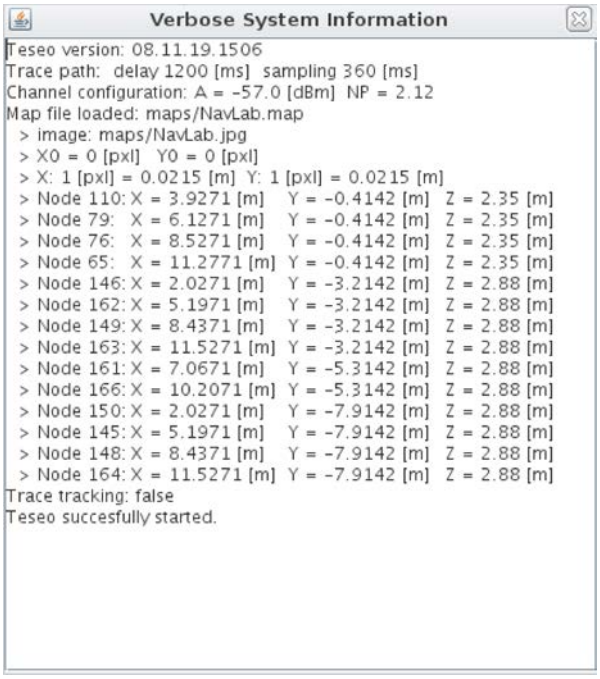

Fig. 11 screenshot of the VSI during the use of Teseo client

- *Init of client-node communication.*
  It is set to *serial* the type of packet source, and it is chosen a default serial port, the USB0 (*/dev/ttyUSB0*) with its baud rate, which is $115200$ baud for the Tmote Sky. The resulting COM port is labeled with the syntax *serial@/dev/ttyUSB0:tmote*.
- *Init of tracing settings.*
  Time values $T_u$ and $\tau_u \propto$ *AUDIT_TIME* are set to $T_u = 360$ ms and $\tau_u = 1200$ ms. These values, editable by the menu *Settings* → *Trace*, pose some temporal constraints on the visualization of the trajectory of the mobile node in the **MapPanel**.
- *Init of channel parameters.*
  It is instantiated the transmission channel defined by the class **Channel**, setting the $\beta$ and $\gamma$ parameters with two default values that are merely suggestive: $\beta = -39.7$ dBm and$\gamma = 3.04$. It is also set to $P_j^{tx} = 0$ dBm the transmission power of each node $j$ at the distance of 1 m.
- *Init of the map.*
  It is Initialized the graphic environment in which the mobile node is displayed, by invoking the method *initEnvironment(String)* of the class **MapPanel**, that receives as input parameters one of the maps stored in the *maps* directory of the package *teseo*.

After the initialization phase, the frame remains in an idle state, as long as the user not only connects the client to the mobile node but also starts the node. Defining the input source to the client, via the control panel, it is possible to start the mobile node by pressing the *Start* button. Doing so, the **ActionEvent** of the **JButton** calls the routine *start()*, which establishes a connection with the mobile node, if it is not been done before, by calling the method *connect(String)*. This method creates an object **PhoenixSource** to automate both the reading and the dispatching of packages and the restarting of the communication port. The **PhoenixSource** is coupled to an object **MoteIF** which provides an interface Java at the application level to receive messages from and send messages to the Tmote Sky. At this point, the **JFrame** is registered as a **MessageListeners** of the **MoteIF** for each of the types of messages **DataMsg**, **MoteCtrlMsg**, **PingClientMsg**. If the connection is successful, the command,START, is forwarded to the mobile node using the function *send(int, Message)* of object **MoteIF**.

The button relegated to the *Stop* of the mobile node behaves similarly: at the pressure of the corresponding **JButton**, the **ActionEvent** calls the routine *stop()*, which immediately forwards the command *STOP* to the mobile node, after verifying that Teseo is connected to it. Then, if the mobile node is successfully stopped, the routine disconnects the client, calling the method *disconnect(String)* which unregisters the listeners of the messages and executes the *shutdown()* of the **PhoenixSource**. From the moment the mobile node is no longer in the state *DO_NOTHING*, the frame becomes sensitive to receive messages transmitted via USB (serial) from the mobile node. The *message_t* received are handled by the synchronized method *messageReceived(int, Message)*, which performs certain operations depending on the type of the received message. If it is a **PingClientMsg** and if it is the first one of this type that the client has received, the frame:

- gains knowledge of the ID of the mobile node with whom the client is connected and it stores its frequency and transmission power;
- synchronizes itself with the mobile node. To do this it is instantiated a **Timer**, which schedules the execution of a **EstimateTimerTask** at a fixed rate of$T_c =$ *AUDIT_TIME* ms. **EstimateTimerTask** is a subclass of the class **TimerTask** and it implements the interface **Runnable**: when the *AUDIT_TIME* ms are passed, the method *run()* of **TimerEstimate** is invoked, which calls the method *estimate2D()* of class **Estimation**, global variable of the frame.

Then the method ends by updating the counter of the steps performed by the mobile node and, if at least one **DataMsg** is not yet arrived, it resets the **HashMap<Integer, Node>** of the **MapPanel** class, related to the fixed nodes that formed the group of nodes used by the mobile node in the previous estimate. If it is a **DataMsg** and if it is the first one of this type that the client has received since the last position estimation executed, the frame resets the **HashMap<Integer, Node>** of the **MapPanel**. Then, if the fixed node to which the **DataMsg** belongs is present in the map, it is added, with his ID, to the **HashMap<Integer, Node>** of the **MapPanel** and its coordinates are added into the **Vector<Coordinate2D>** of the **Estimate** together with the measure of the RSS that is put in column of the **Vector<Integer>** of the class **Estimate**. The method *messageReceived int, Message)*, as mentioned, continues to discriminate messages for $T_c$ ms, and then decreed the beginning of the process of mobile node position estimation, assigned to the class **Estimate**. Before the timer expires, the client must be able to form the set of fixed nodes assigned to the current step, assuming that the mobile node is inside a communication range that allows him to communicate with a non-empty group of fixed nodes, in order to allow the MAT algorithm to make an estimate that is not the simple evolution of the state of an open-loop system. For a deeper understanding of the functioning of the client will now be outlined, one by one, the classes **Channel**, **Estimate**, **MapPanel**, **VariantExtendedKalmanFilter** that are those that most characterize the MAT algorithm.

### *A. Channel*

This class implements the channel model presented in Sec. 2. The constructor requests to set the transmission power $P_j^{tx}$ of all fixed nodes, the attenuation $\beta$ of the channel and the loss factor $\gamma$. Method *getPower(double)* returns the value of power $P_{ij}$, i.e. the power of the mobile node $i$ function of the distance from the fixed node $j$. The $\mathbb{R}^2$ distance between two nodes is given by the method *get2Dnorm(double, double, double, double)*. Essential to compute the estimation is the method *getDerivativesPower(double, double, double)*, that returns the elements of (6).

### *B. VariantExtendedKalmanFilter*

This class hold the model and dynamic of the EKF, implementing Alg. 1. It uses the JAMA (*JAva MAtrix package*) library [22], version 1.0.2, a linear algebra package that provides user-level classes for constructing and manipulating real, dense matrices. The constructor of the class builds matrix $\mathrm{A}$ of the state model, method *setInitialCondition()* fixes the initial conditions, as they has been defined in Sec. 3. Instead, method *update(double[][], double[][], Channel)* is designed to implement Alg. 1.

### *C. Estimate*

This class is focused on the synchronized method *estimation2D()*, which is divided into the following sequential operations:

- copy in different arrays the values stored by the vectors **Vector<Coordinate2D>** and **Vector<Integer>**, which are passed to method *update(double[][], double[][], Channel)* of **VariantExtendedKalmanFilter2D**. **Vector<Integer>** stores the measures of RSS made by the mobile node in reception of **DataMsg** messages from fixed nodes, while **Vector<Coordinate2D>** stores the corresponding $\mathbb{R}^2$ positions of the fixed nodes;
- execution of the update routine own by the class **VariantExtendedKalmanFilter2D**. This routine, core of the MAT algorithm, returns, in a period of 1 ms circa, the estimation of the $\mathbb{R}^2$ position of the mobile node. This values is then stored in a global **Coordinate2D** variable of **Estimation** class;
- management of mobile node position. It is saved in the object **Node** of **MapPanel**; then it is added both to **Vector<Node>**, that collects all the estimated positions that has to be logged, and to **Vector<Node>** of tracing option, under some constraints given by the delay $\tau_u$ and the sampling time $T_u$. Moreover, some variable are reset and the *estimate2D()* method terminates clearing vectors **Vector<Coordinate2D>** and **Vector<Integer>** and notifying to the frame that the estimation process is finished.

At this point the **JFrame** is ready to wait a new **PingClientMsg** message, to be followed by other **DataMsg** message used to produce a new estimate of the position of the mobile node.

### *D. MapPanel*

**MapPanel** is a sub class of the **JPanel** which aim is to coordinate the visualization of the graphical environment of the map. Teseo holds a global instance of **MapPanel**, initialized invoking the routine *initEnvironment(String)*. The first activity of the **MapPanel** is to upload both the image file of the default map and the locations of fixed nodes, deployed in advance within the various locations on the map. To simplify this configuration step it has been implemented a parser that reads the settings directly from text configuration files identified by the extension *.map*. The parsing is performed by the method *parseMapFile(String)* which is passed as a parameter the name of the configuration file, contained in the subdirectory */maps* of package *teseo*. The successful completion of parsing operation depends on the following rules:

- first line of the file has to contain the name of image file of the map. This file has *.jpg* extension, and its dimension must be equal to $640\mathrm{x}480$ pixels;
- second line has to point the position of the origin of the 2-dimensional referral system, used to measure the position of the nodes in the environment. The unit of the origin should be in pixels;
- third line has to indicate the length in meters of a screen pixel. This is an essential factor of scale in order to display properly the position of the nodes (fixed or mobile) in the GUI;

- following lines have to list the details of each fixed node. The details are the ID, the abscissa, the ordinate and the z-axis of the nodes. The values are separated by a comma and the unit of the coordinates is in meters.

Lines of */maps* file are progressively enumerated, excluding empty lines and comments. The file scan is performed using a simple text scanner of the Java standard library which can parse primitive data types and strings using regular expressions. A **Scanner** separates the input into various tokens distinguished by a delimiter pattern, which by default is the whitespace character. Nodes are saved in a **HashMap<Integer, Node>**. Indexing is performed using as key the node IDs, which are intrinsically unique. If necessary, to access the contents of the entire map **HashMap<Integer, Node>**, it is possible to use an **Iterator** on the **Set** of the keys. This set is accessible in one step through the method *keySet()* of the class **Map<Integer, Node>**. After the completion of the parsing, the method *loadImage(String)* loads the icons associated to the mobile node, the fixed nodes and the virtual origin of the axes. The mobile node is marked by a green rectangle, while the fixed nodes, at each $k$ step of the MAT algorithm, are blue or red rectangle. The color of the fixed nodes depends on whether or not the nodes belongs to the set chosen by the mobile node to estimate the position in the time interval$[(k-1)T_c\ kT_c]$. If a fixed node is selected it turns to blue, hence it is red colored. The origin of the reference system, whose display is optional, is an olive green viewfinder. Assuming that the loading of some images is not successful, it is expected to replace the images with a rectangle of the class **Graphics**. The last action of the initialization, afterwards creating **Vector<Node>** of the *Trace Path* functionality, it is the start of the refreshing **Thread** of the **JPanel**, whom sampling time is given by the constant *REFRESH_TIME*. The refresh repeatedly calls the method *repaint()* which in turn invokes *paintComponent(Graphics)*. This one draws the map and the origin; the fixed nodes, iterating *drawAnchorNode(Node, Graphics, boolean)*, if it is checked the **JCheckBoxMenuItem** *Beacons*; the mobile node, *drawMobileNode(Graphics)*; the trace of the path of the mobile node, if requested, obtained with a linear interpolation of the positions. Lines are drawn with the method *drawThickLine(Graphics, int, int, int, int, int, Color)*. The positions are taken in chronological order from a **Vector<Node>** thanks to an iteration on a **Enumeration** of the elements of the vector. Furthermore, starting from a certain length of track, more than *MIN_TRACE_SIZE*, the history of the trace begins to be erased, giving to the path a snake effect.

## VII. Simulations with Experimental Setup

To validate the algorithm described in Sec. 3 simulations have been performed on the base of the network data derived from the WSAN installed in the Department of Information Engineering (DEI) of the University of Padova [5]; the testbed considered (a portion of the mentioned WSAN) is depicted in Fig. 12 and comprises $12$ Tmote Sky [10] whose Chipcon CC2420 radio has an accuracy of $6$ dBm.

Here, the agents have a distance of about $4$ meters from each other on an almost regular triangular grid of $15 \times 10$ m$^2$. This testbed is partially unstructured with laboratory/office furniture and equipment, and three partition walls separate two rooms with an hallway. The agents communicate only through the wireless channel and. Access points are also present in the environment, hence the testbed is subject to a reasonable level of interference and electromagnetic noise.

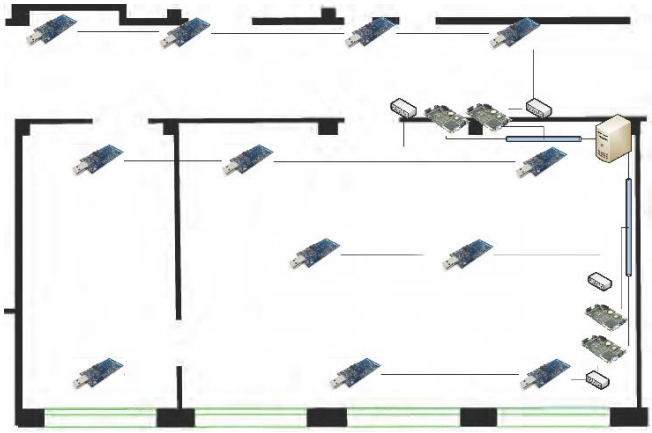

Fig. 12 architecture of the testbed, covering an area of about 150 m$^2$. The agents communicate only through the wireless channel

All Tmote Sky agents, in groups of up to four elements, are connected via USB (serial) hubs that provide power supply and allow to collect log data for debugging intents. The agents are also connected to embedded computers that act as gateways. These mini PCs are processing units which interact with the programming of the agents and they are connected via Ethernet to a central server from which to monitor, manage and check the entire WSN.

For the estimation of the channel parameters $\beta$, $\gamma$ in (3) the least-square method in [19], which is a distributed version of [20], has been adopted. The results ($\gamma = 2.04$, $\beta = -41.69$ dBm, $\sigma^2 = 7.57$ m$^2$) provide the model in Fig. 13.

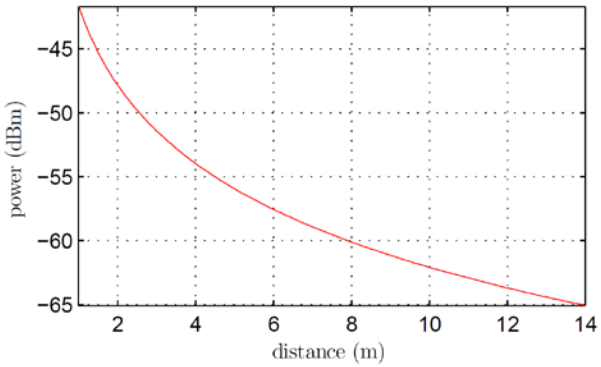


Fig. 13 power model $P = \beta - \gamma \log_{10} d$, as function of distance $d$. Notice that the plot is limited to distances below $15$ meters, since it is not worth to consider larger intervals

The packet loss probability in Fig. 14, equal for each agent, is obtained as a least-square interpolation of experimental data collected in the testbed of DEI.

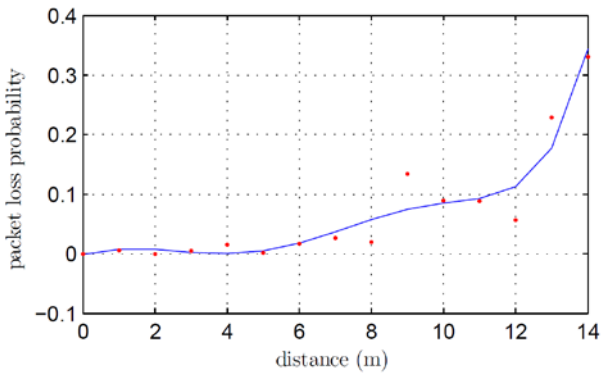


Fig. 14 packet loss probability. The red dots are samples computed on experimental data; the blue line is their least-square interpolation

The movement of an agent is simulated through a random walk model

$$\boldsymbol{\xi}(k+1) = \begin{bmatrix} 1 & 0.1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0.1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \boldsymbol{\xi}(k) + \mathbf{w}(k)$$

$$\boldsymbol{\psi}(k) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \boldsymbol{\xi}(k) + \mathbf{v}(k)$$

where, $\xi_3(0) \sim \mathcal{U}[0, 10]$, $\xi_2(0) = \xi_4(0) = 0$ and the variances of model and measure noise $\mathbf{w}(k)$ and $\mathbf{v}(k)$ are respectively given by:

$$Q = 9.4 \begin{bmatrix} 0.01 & 0.1 & 0 & 0 \\ 0.1 & 1 & 0 & 0 \\ 0 & 0 & 0.01 & 0.1 \\ 0 & 0 & 0.1 & 1 \end{bmatrix} \quad R = 0.0315 \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

*A. Performance evaluation*

In general, the performances of any tracking algorithm depend on different factors, such as density and connectivity of the beacons, computation and communication costs, fault tolerance and robustness. In Fig. 15, the position estimation error $||\widehat{\boldsymbol{\xi}} - \boldsymbol{\xi}||_2$ is plotted for different algorithm parameters, as a criteria to evaluate the goodness of the tracking algorithm.

Interestingly, the value of $C$ (maximum number of RSS data that each agent collects from neighbors to average the received power), over a certain threshold, does not affect significantly the position estimate, while the promptness of the system slows down increasing $C$. The system behaves similarly as for the bound on the received power, and increasing $RSS_{bound}$ (the minimum power level acceptable for node-to-node distance estimation) would lower the number of useful signals in the localization process. Finally, increasing the measurement noise variance $\sigma_v^2$, worsen the performance, as expected.

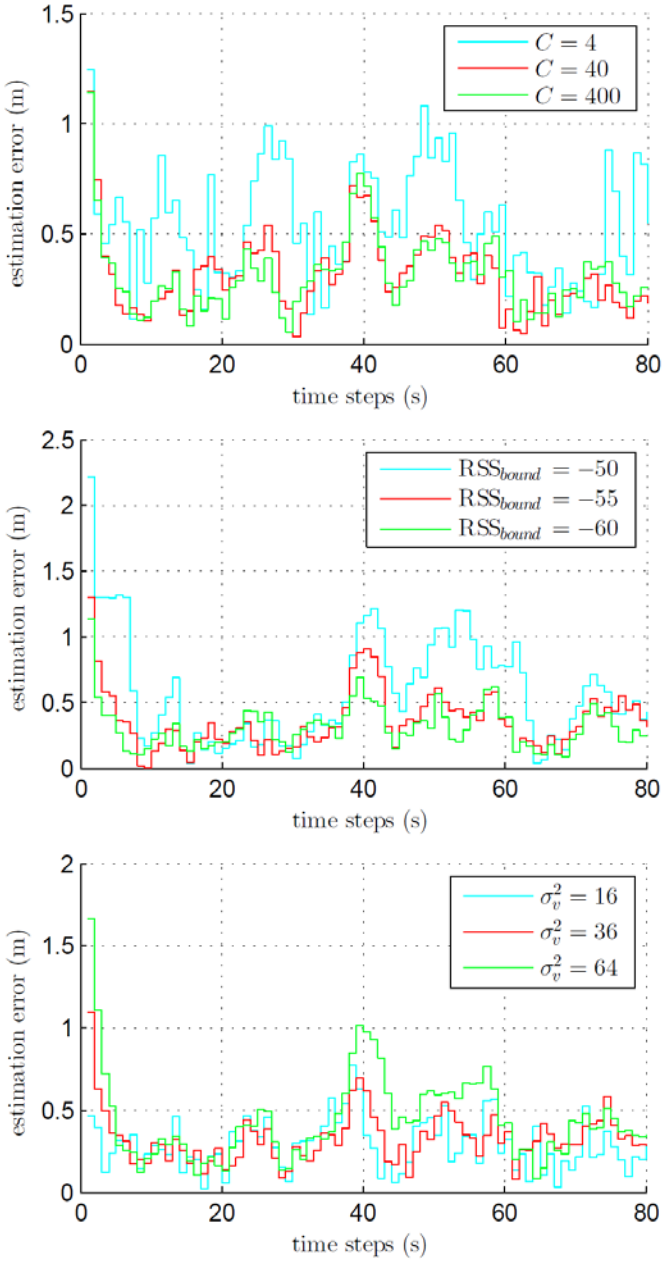


Fig. 15 Estimation errors for different simulation parameters

If the extended version of the EKF has become necessary to deal with the non-linearity of the system, the use of an Unscented Kalman Filter (UKF) or a Sequential Monte Carlo (SMC) method has not be considered since these two approaches are proven to not improve significantly the performance in terms of localization accuracy. In fact, the SMC, which is in general a better solution than a UKF [23], tends to outperform the Kalman as the localization errors increase and it cannot considerably filter the non-Gaussian components.

## VIII. Conclusion

In this work, an application for multi-agent tracking in wireless networks, with emphasis on the software design and the code implementation, is presented. The application employs a RF-channel model to estimate the distance among agents belonging to the WSAN. To mitigate the nuisances induced by the not perfect wireless communication, by the implementation in an unknown and unstructured environment, and by the presence of noisy measurements, an EKF is designed to provide corrected estimates of the mobile agent positions. Moreover, attention has been posed on the timings among the events occurring within the agent and the synchronization with the other peers of the network, to ensure the correct sequence and completion of the tracking procedure. Simulations and experiments on a real testbed validate the goodness of the approach and assess it is suitable for a real time implementation on embedded devices.

## Acknowledgment

The authors gratefully acknowledge Eng. Fabio Maran for his support to develop the simulation environment and Massimo Marra to collect the experimental data.

## References

[1] K. Romer, F. Mattern, "The design space of wireless sensor networks", *IEEE Wireless Communications*, vol. 11, no. 6, 2004, pp. 54–61.
[2] L. M. Oliveira, J. J. Rodrigues, "Wireless sensor networks: a survey on environmental monitoring", *Journal of Communications*, vol. 6, no. 2, 2011, pp. 143–151.
[3] Z. Bojkovic, B. Bakmaz, "A survey on wireless sensor networks deployment", *WSEAS Transactions on Communications*, vol. 7, no. 12, 2008, pp. 1172–1181.
[4] S. M. Torabi, M. A. Samadian, "Covering of problem in wireless sensor networks", *WSEAS Int. Conf. on Telecommunications and informatics (TELEINFO09)*, pp. 88-94
[5] P. Casari, A. Castellani, A. Cenedese, C. Lora, M. Rossi, L. Schenato, M. Zorzi, "The wireless sensor networks for city-wide ambient intelligence (WISE-WAI) project", *Sensors*, vol. 9, 2009, pp. 4056–4082.
[6] A. Deshpande, C. Guestrin, S. R. Madden, "Resource-aware wireless sensor-actuator networks", *IEEE Data Engineering*, ch. 28.
[7] V. Gupta, R. Pandey, "Data fusion and topology control in wireless sensor networks", *WSEAS Transactions on Signal Processing*, vol. 4, no. 4, 2008, pp. 150–172.
[8] M. Hefeeda, M. Bagheri, "Forest fire modeling and early detection using wireless sensor networks", *Ad Hoc & Sensor Wireless Networks*, vol. 7, no. 3-4, 2009, pp. 169–224.
[9] A. Cenedese, G. Ortolan, M. Bertinato, "Low density wireless sensors networks for localization and tracking in critical environments", *IEEE Transactions on Vehicular Technology*, vol. 59, 2010, pp. 2951–2962.
[10] Moteiv, Tmotesky. (2006, May 10). [Online]. Available: www.snm.ethz.ch/Projects/TmoteSky.
[11] P. Lewis, *Tinyos programming*, Oct. 2006.
[12] D. Gay, P. Lewis, R. von Behren, et al., "The NesC language: a holistic approach to network embedded systems", PLDI, 2003.
[13] F. Zanella, Teseo. (2006) [Online]. Available: sourceforge.net/projects/teseus.
[14] I. Amundson, X. D. Koutsoukos, "A survey on localization for mobile wireless sensor networks", *Int. Conf. on mobile entity localization and tracking in GPS-less environments*, 2009, pp. 235–254.
[15] K. Lorincz, M. Welsh, "MoteTrack: a robust, decentralized approach to RF-based location tracking", *Personal and Ubiquitous Computing*, vol. 11, no. 6, 2006, pp. 489–503.
[16] N. Patwari, *Location estimation in sensor networks*, Ph.D. thesis, University of Michigan, 2005.
[17] D. Simon, *Optimal State Estimation: Kalman, H Infinity, and Nonlinear Approaches*, 1st Edition, Wiley-Interscience, 2006.
[18] W. Su, M. Alzaghal "Wireless sensor network: channel propagation measurements and comparison with simulation", *WSEAS Int. Conf. on Computers (ICCOMP07)*, 2007, pp. 208–213.
[19] A. Cenedese, F. Zanella, "Channel model identification in wireless sensor networks using a fully distributed consensus algorithm", University of Padova, 2012, technical report.
[20] S. Bolognani, S. Del Favero, L. Schenato, D. Varagnolo, "Consensus-based distributed sensor calibration and least-square parameter identification in WSNs", *Int. Journal of Robust and Nonlinear Control*, vol. 20, no. 2, 2010, pp. 176–193.
[21] A. Boukerche, H. Oliveira, E. Nakamura, A. Loureiro, "Localization systems for wireless sensor networks", *IEEE Wireless Communications*, vol. 14, no. 6, 2007, pp. 6 –12.
[22] J. Hicklin, C. Moler, P. Webb, R. F. Boisvert, B. M. R. Pozo, K. Remington, Jama - java matrix package (2005). [Online]. Available: http://math.nist.gov/javanumerics/jama.
[23] K.-C. Lee, A. Oka, E. Pollakis, L. Lampe, "A comparison between unscented kalman filtering and particle filtering for rssi-based tracking", *Workshop on Positioning Navigation and Communication* (WPNC), 2010, pp. 157-163.

**Filippo Zanella** was born in Valdobbiadene (Treviso, Italy) in 1983. He received his B.S. degree and M.S. degree in Automation Engineering from the University of Padova, Padova, Italy, in 2005 and 2008 respectively.

His research interests are in the areas of wireless cameras/sensors networks and mobile networks with emphasis on distributed control, estimation and optimization.

He is currently a Ph.D. Candidate at the Department of Information Engineering at the University of Padova. He has been a Visiting Student Researcher at UC Berkeley in 2011 and at UC Santa Barbara in 2012. Dr. Zanella is Member of IEEE since 2006 and he has been Staff Member of the IEEE Student Branch of the University of Padova from 2006 to 2008.

**Angelo Cenedese** was born in Treviso (Italy) in 1972. He received the Laurea degree in 1999 and the Ph.D. degree in 2004, both from the University of Padova, Padova, Italy.

His research interests are in the fields of modeling, control theory and its applications, active vision, sensor and actuator networks, with particular attention to environmental monitoring and control, and camera networks.

He is currently an Assistant Professor with the Department of Information Engineering, University of Padova. He has been involved in European Union projects on control and diagnostics of nuclear fusion devices, on methodologies for adaptive optics systems, and on estimation and control problems in distributed networked systems. He has coauthored around 70 papers.