Multi-agent tracking in wireless sensor networks: implementation

FILIPPO ZANELLA
University of Padova
Department of Information Engineering
Via Gradenigo 6/B, 35131 Padova
ITALY
fzanella@dei.unipd.it

ANGELO CENEDESE
University of Padova
Department of Information Engineering
Via Gradenigo 6/B, 35131 Padova
ITALY
angelo.cenedese@unipd.it

Abstract: In this work the design and implementation of an application to track multiple agents in an indoor Wireless Sensor Actor Network is proposed. The adopted embedded hardware for the network nodes is the TMOTE™ SKY, 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 Wireless Sensor Actor Networks. These devices are based on the TINYOS operative system and are programmed in NESC, a C-derived language specifically developed for embedded systems. NESC has become indispensable for low-level management of individual agents while JAVA was chosen to provide the user with a simple and intuitive graphical interface to show and coordinate the tracking.

Key–Words: wireless sensor network, embedded systems, TINYOS, NESC

1 Introduction

In recent years, the employment of Wireless Sensor Actor Networks (WSANs) for gathering data from the environment have been increasingly envisaged for building management systems and environment control [1][2][3], thanks to their versatility of use, easiness of deployment, pervasiveness of data, adaptability to system and environment variations [4].

This revolution 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 [5], 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 [6] and are programmed in NESC [7], a C-derived language specifically developed for embedded systems.

In this companion paper we describe the implementation stage of a wireless network for RF-based localization and tracking, where the aspects related to the mathematical model and algorithms has been presented and discussed in [8]; to briefly recall the context, we consider the scenario where a set of mobile devices (i.e. mobile nodes) is moving within a network of fixed nodes (i.e. fixed nodes), with which they communicate through the RF-channel exchanging information on the surrounding.

The implementation of the algorithm in this framework appears 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 [9], distributed under the GNU General Public License.

2 Software design

A set of indexed mobile nodes $M = \{m_1, \ldots, m_M\} \subseteq \mathbb{N}$ moves within a network of indexed fixed nodes $F = \{f_1, \ldots, f_F\} \subseteq \mathbb{N}$, each node running a TINYOS module and communicating via wireless, assuming the parameters of the radio channel as known [10]. 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 [8] 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 \( C_m \) 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 \( D M_{max} \) messages every \( T_I \) 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 \( C_m \);

5. \( C_m \) stores the messages and every \( T_c \) ms estimates the position of \( m_i \), showing it in a GUI.

Fig. 1 outlines the schema of MAT scheduling, for a complete cycle of the algorithm of \( T_s = \text{TIMER\_STEP} \) ms. It compares with the same time scale the operating modes of the fixed nodes, the mobile node and the client. Scheme of Fig. 1, 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.

3 Implementation: NesC for nodes

Four message types are defined to exchange information among different devices (Fig. 2):

- **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 \( C_m \) to \( m_i \);
- **ping_client_msg**, to ping the clients. It is used by \( m_i \) to inform \( C_m \) that a MAT is ready to start and to sent configuration settings. This message is sent via USB from \( m_i \) to \( C_m \);
- **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 \( C_m \), enabling the position estimate. This message is broadcast via radio by \( \{ f_i \} \) to \( m_i \) and via USB by \( m_i \) to \( C_m \).

![Figure 2: 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.](image)

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 addiction, \( m_i \) is enabled/disabled by \( C_m \) through the following commands:

- **START_MN**: starts mobile node;
- **STOP_MN**: stops mobile node.
3.1 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.

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 $C_m$. 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.

Clock Step

When $m_i$ receives a `START_MN` from $C_{m_i}$, it starts the timer `ClockStep` that every $T_s = \text{TIMER\_STEP} \text{ ms}$ 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.

Clock Send Ping Client

When $T_p = \text{TIMER\_SEND\_PING\_CLIENT} \text{ ms}$ elapse, $C_{m_i}$ is repeatedly informed of the start of the MAT process, for a number of times equals to $PCM_{max} = \text{MAX\_PING\_CLIENT\_MSG}$. 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`.

Clock Send Ping Node

When $T_n = \text{TIMER\_SEND\_PING\_NODE} \text{ ms}$ are elapsed, task `sendPingNodeMsg()`, periodically posted by the timer, broadcasts $PNM_{max} = \text{MAX\_PING\_NODE\_MSG}$ 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 $\{f\}$ in range, it moves to the `AUDIT\_NODE` states and stops the
timer ClockSendPingClient. Then it waits to receive data_msg messages.

**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 `sendDataMsg()`, 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.

### 3.2 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.

**Boot**
When one fixed node \(f_i\) turns on, TINYOS starts the boot sequence. In the function `booted()` peripherals 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.

**Receive ping_node_msg**
When \(f_i\) receives a first ping_node_msg message from a mobile node \(m_i\), identified by a unique `ID[k]`, \(k \in [1, PNM_{max}]\), it starts the timer `TimeToSend` that every \(T_i = \text{TIMER_TRANSMISSION} \text{ ms} \) 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:

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

where \(T_s\) and \(T_n\) are the times previously defined in Subsec. 3.1. 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 [11]. This bound in the transmission of the data_msg message forces \(f_i\) to move to the state IDLE after \(T_i \cdot DM_{max}\) ms, here remaining unless it receives other ping_node_msg by some moving \(m_i\) present in the environment.

**Clock Send Data Node**
When \(T_i\) 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.

### 4 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.

As for the former issue, we refer to the companion paper [8], while for the latter part we briefly present an overview hereafter.

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:
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.

The `command console` 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. In Fig. 4 there are shown the flow charts of the routines `start()` and `stop()`.

The `informative panel`, displays the numerical value of the 2D 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.

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$^\text{TM}$ 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.

From the moment the mobile node is no longer in the state `DO NOTHING`, the frame becomes sensitive to receive messages transmitted via USB.

![Figure 3: View of the GUI TESEO.](image)

![Figure 4: Flowchart of the start/stop of the mobile node and the client.](image)
(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 \( T_e = \text{AUDIT\_TIME} \) 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 its 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 and its coordinates are added into the MapPanel received, the frame:

- the 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.

5 Conclusions

In this work, we presented the software design and the code implementation of a multi-agent tracking algorithm envisaged for WSANs. We remark how such issues are of paramount importance when dealing with embedded devices, because of the limited resources available. In particular, attention needs to be posed on the timings among the events occurring within the agent and the synchronization with the other peers of the network, thus justifying the adopted state machine approach to ensure the correct sequence and completion of the procedure.

References: