Documentation - Urban HCI
Distributed Klick/Flickable Interfaces for Public Spaces

Daniel Pollack
Medieninformatik Bc.
daniel.pollack@uni-weimar.de

Jenny Gonzalez
Computer Science and Media
M.Sc. student
jenny.carolina.gonzalez.acuna@uni-weimar.de

Ingo Schäfer
Medieninformatik Bc.
ingo.schaefer@uni-weimar.de

1. ABSTRACT

In this work, we explain the implementation of a distributed interface for public spaces where feet play the main role. Users interact with the interface by pushing or kicking the individual modules that constitute the system and each module reacts to the stimulus differently;that is, each one of them expresses a different *personality.* In this system, a server orchestrates all the modules

Each individual module is made up of a sensor and an actuator node; the former gathers information and sends data to the server while the latter receives data from the server and reacts to it accordingly. The server is constituted by two entities: a computer with a Java-based server and a server node, both of them interconnected. With the server node as the core, all actuator and sensor nodes establish a wireless network with a star topology, where the information is mainly transferred between a node and the server. The Java-Server manages all network nodes depending on the recollected data, and provides an interface to allow users the configuration of different settings in the network. With the interface, users can create their own setups and get a quick and complete overview of the network status. The Java-server determines the personality of each entity.

We present summaries of related projects and then describe the technical development of our foot-based interface, its final structure and how it works.

2. RELATED WORK

The goal of this project was to create a new kickable and flickable interface for outdoors. For this, we first did some research about interactive systems that were designed for public spaces and interfaces that focused on feet interaction. Here, we present some interesting systems that inspired us during the creative phase of our work.

Seitinger, S. et al. [16] created the 'Light bodies'(see Figure 1): mobile and portable, hand-held lights that respond to audio and vibration input. Street lighting has changed over time and its relation to individuals has been also altered as a result. In the middle ages, each person used its own lantern. Then, in the beginning of 18th century, cities like Paris started to illuminate their streets at night; personal lanterns were not longer required. 'Light bodies' is an attempt to bring back the lost connection between lanterns and people and a medium to investigate the relationships between (urban) spaces, light and responsive, hand-held lights.'Light bodies' enabled people to directly and indirectly influence their personal lightscape.

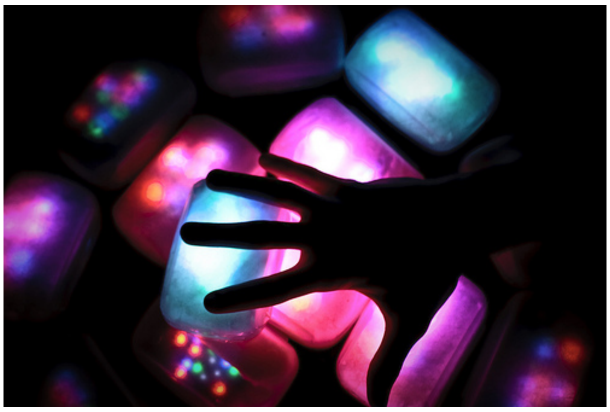


Figure 1: Light bodies

In another related project, Seitinger, S. et al. [15] designed and implemented the 'Urban pixels': physically instantiated pixels that enable flexible, reconfigurable, unbounded, low-resolution, and responsive urban displays (see Figure 2). 'Urban Pixels' are nodes in a wireless network of physical pixels for urban spaces. Each pixel unit includes a microcontroller, RF transceiver, LED module (ten bright, white LEDs), rechargeable Li-Ion battery pack, IR sensor and renewable energy source such as photo-voltaic cells. Users can interact with the 'urban pixels' by placing them spontaneously, by triggering sensors or by sending messages from their mobile devices.

With this design, Seitinger, S. et al. [15] wants to present an opposite alternative to the majority of displays that are inflexible, flat, bounded, high-resolution and unresponsive.

As an example of an interface where feet play the main role,

Figure 2: Urban-pixels

Paelke et al. [11] implemented a system for foot-based mobile interaction, which uses a camera of current video capable mobile devices to detect motion and position of the user's foot to effect the input that is required for simple games like *Pong*, *Break-out* or *soccer*. Figure 3 shows an scenario where a user interacts with its mobile phone by moving one feet.

Figure 3: Mobile phone game and user interacting with it by moving a foot

3. CONCEPT

3.1 Basic Concept

Feet are used in many real world tasks together with the rest of the body, but in computer environments they are almost completely put aside as an interaction possibility [12]. For instance, Ubicomp researches have developed different types of interfaces for public spaces where users' hands are the protagonists and feet are partly or completely neglected.

The basic concept of this project is a feet-based interface whose **individual modules** react when pushed or kicked. For this purpose, modules have to be highly **mobile** to yield a flexible interaction with the user. Another desired aspect is **expandability**: the envisioned system will allow the integration of new modules and features. Finally, a module is expected to detect if a another module comes up against it. In other words, another feature in the envisioned system is the **interaction between modules**.

3.2 Hardware Concept

To achieve **modularity** and **mobility**, a wireless communication platform is fundamental. Panstamps [14] are small boards intended for implementing custom wireless networks. In their boards, Panstamps have an integrated transceiver, the CC1101 [17], that makes the wireless communication possible.

Due to Panstamps' open source nature, documentation and information about already implemented projects with them is at hand for everyone on the internet; this represents a very important advantage over other wireless products in the market. Furthermore, Panstamps' core is the the ATMEGA 328 [5]; therefore, they are entirely programmable from the Arduino IDE, whose community is increasing each day.

For all these reasons, our envisioned system is based on Panstamps. As depicted in Figure 4, each module in our system is constituted by a sensor and actuator node, both having a Panstamp as their nucleus. The server node has also a Panstamp as its main component, which communicates wirelessly with the other modules' panstamps.

The Java Server gets the data recollected by the server node, process it and gives a response to the server node that has to be sent to the appropriate node in the network. The feedback provided by the Java Server depends entirely on the *personality* of each device (module).

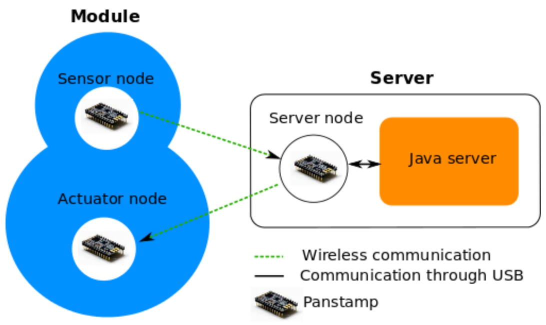


Figure 4: Interface's server and a module

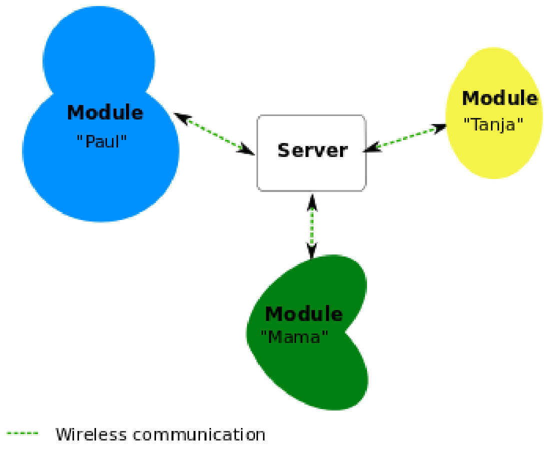


Figure 5: Three different modules in the network and the server

To detect if they have been kicked or pushed, modules need to sense their own movement; therefore, an accelerometer is connected to the sensor node of each module as shown by Figure 6. We used the ADXL335 [2], a small accelerometer that measures acceleration in each of the cardinal directions (x,y and z).

Each module has a led strand constituted by led-and-controller pairs. To produce a certain light pattern, the actuator node, which is connected to the strand (see Figure 7), sends information to each controller to indicate whether its led should be on or off and what color has to be displayed. Led patterns play an important role in the interface because each module expresses its **personality** through led patterns.

The **interaction between modules** is based on proximity; that is, a module triggers a reaction in other module when it is close to it. In our implementation, the closeness between two modules is determined by measuring the RSSI of broadcasted messages in the network. The RSSI is a measurement of the power present in a received radio signal. The longer the distance travelled by a radio wave the bigger its intensity loss.

All the elements in a module are powered by a 11 volts Li-Po battery. This battery should not be over-discharged, for this reduces drastically it's life. Therefore, sensor nodes have an integrated voltage-monitoring circuit (see Figure 6) that detects the low-battery state.

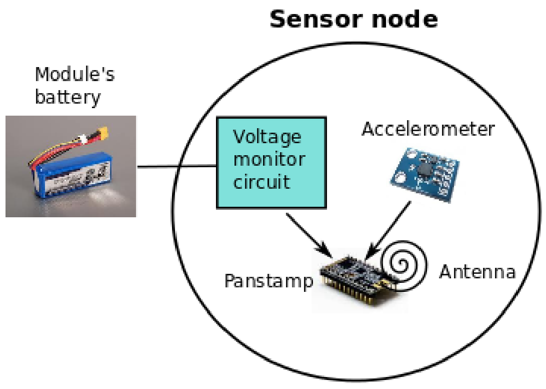


Figure 6: Sensor node elements

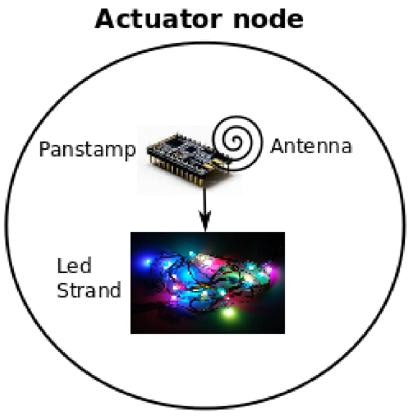


Figure 7: Actuator node elements

3.3 Software Concept

3.3.1 Java server

The Java Server is the heart of the interface; it is the orchestrator of all nodes in the system's network. The server node transfers all the data gathered in the network to the Java Server, which produces a response according to each module's personality and sends it to the intended actuator nodes through the server node (See Figure 4).

The Java server's main functions are:

- to provide an interface that allows the configuration of modules' personalities
- to analyse data transmitted by the sensor nodes
- to generate responses to the sensed stimulus by sending commands to the actuator nodes
- to store modules' personalities

3.3.2 Driver

Since we used the Arduino IDE to program the Panstamps, all the nodes' drivers in the network were written in C++. These drivers were designed to allow **expandability**; as a result, new nodes can be added to the network by coping the basic drivers and changing them slightly.

In short, the drivers are in charge of:

- enabling the wireless communication
- managing the communication between different nodes
- handling the information provided by sensors
- operating the led strands
- setting up the communication with the Java Server on the Panstamp' side

4. DEVELOPMENT

In order to find ideas of how to implement 'KickandFlickable' devices/interfaces, we first started to learn about the technological components we had at hand, such as the arduino. For example, we made some of the start-up projects published in the lady ada tutorial [9].

We learned how to use basic components like LEDs and tilt-sensors. Later, we worked with more complicated components such as the hall effect sensors (See section 8). Meanwhile, we did experiments with XBees[6] and found that its signal range was relatively short for our purposes (the maximum range we obtained was approximately 5 meters). Thus, we decided to start working with Panstamps [14], which are a compact wireless versions of the Arduino devices.

After exploring different options, we chose the accelerometer for sensing movement in the modules because its output signal is not binary (as the tilt sensor) but continuous and this property gives more flexibility, since not only two but many states can be defined with its data.

For making the detection between modules possible, we decided to take advantage of the RSSI of the signals produced by the panstamps. Since precision was not a critical factor in our interface, RSSI measurements suited our technical needs.

To give feedback to the users when they kicked or slightly moved the modules, we decided to use multi color led strands. Light stimulus catches easily user's attention, specially in the night. Moreover, led strands allow the implementation of different color patterns and as a result, different light responses can be given to the users depending on the different ways the move the modules with their feet.

5. THE JAVA SERVER

5.1 First development steps

As explained in the Software Concept section, the Java server receives from the server node all the data gathered in the network, produces a response according to each module's personality and sends the corresponding commands to the intended actuator nodes through the server node.

To develop the Java server, we first divided its envisioned structure into subcomponents and worked with each part at a time. The first subcomponent was a serial port communication system, the second was the graphical user interface. Later, we learned about threads and runnable concepts in Java to implement another subcomponent dedicated to the management of the different nodes in the network.

As a first step, we implemented a basic serial port communication routine for learning and experimenting purposes. For this, we worked with the RxTx-library to get a better understanding of the tools it provided. The small application only wrote information to the serial outputstream and if there were available data coming from the serial port, it read the data and printed them in the console.

After understanding the ground principles of the library, we began to design and implement the Java server.

5.2 Graphical User Interface

The Java Server's User Interface is constituted by two parts: the Main Window and the Configuration Dialog.

5.2.1 Main Window

The Main Window has two sections: the first allows the connection set-up with the panstamp server while the second displays the overall state of each module in the network (see Figure 8). To establish a connection, the user has to select the serial port, which the panstamp is attached to, and the corresponding baud-rate. Once the connection is set up, the "connect" button is substituted by a "disconnect" button. The user can disconnect and connect the panstamp at any time. If the Java server does not detect a device (module) in any of the serial ports, the "connect" button will be unavailable (i.e. grayed out). To prevent the user from constantly restarting the server, we implemented a refresh button, which can be pressed at any time. When a device is finally detected, both options (connect and disconnect) will be available for that device.

In the second section of the Main Window, the overall state of each device (module) in the network is shown in a table (see Figure 8). Among the data displayed are: the name of the module's personality, the direct neighbour, the actual state, the last corresponding time to the server and the battery state.

For this project, we implemented three standard personalities named "Paul" (default personality), "Tanja" and "Mama".

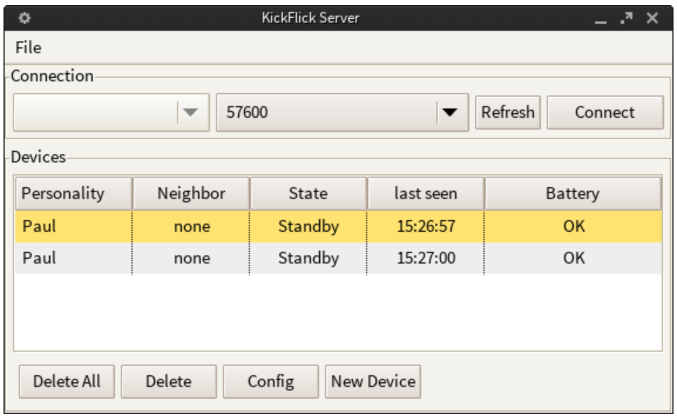
After the start-up and before users interact with the modules, these devices won't have a direct neighbour so the field 'Neighbour' initial value is 'none'. When a module detects a neighbor module, the field 'Neighbour' will display the name of the detected neighbor's personality.

The field 'State' indicates how persistently a module is being kicked. Before users interact with the modules, each device is in "stand-by". After the first interaction, the state will change to "first contact"; then, the following states are "played" and "played (hard)". Each device (module) goes through all possible stages before reaching "played (hard) state; thus, a direct change from "first contact" to "played (hard)" is not possible.

The forth field 'last seen' indicates the time when the last communication between device and server has been.

The last row in the table, 'Battery', indicates the charge of the Li-Po battery and the two possible states are: "OK" and "low battery".

To reflect new detected modules in the network and changes in the devices' status, the information on the table is periodically updated; the table's refresh rate is configurable. The update process is implemented using a timed thread, which is periodically checking the modules' data and altering the shown information accordingly. In case the battery state was set to "low battery" or the last received message was received long ago (when 'last seen' value surpasses a configured threshold), the main window will change the background of the device's personality entry in the table to red to alert the user.



Personality	Neighbor	State	last seen	Battery
Paul	none	Standby	15:26:57	OK
Paul	none	Standby	15:27:00	OK

Figure 8: Main Window

If the server detects a new module in the network, the user can select it in the device-table and open a configuration dialogue by either pressing the configuration button (located on the bottom of the table) or by clicking on the intended table's item. The configuration dialogue will then be opened to allow the configuration of this particular device's personality.

5.2.2 Configuration Dialogue

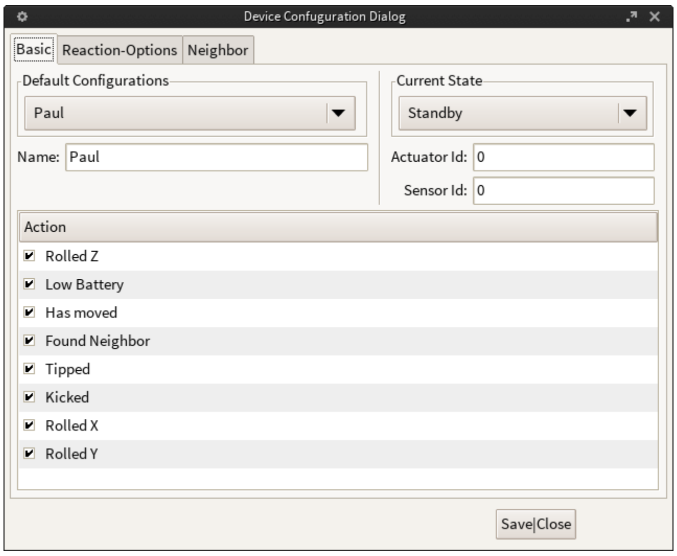
The Configuration Dialogue provides all the information about the selected module's state and allows the configuration of the device's personality.

The user can customize a personality or select one among the predefined personalities. But if the user creates a personality and then selects a predefined one, all the changes made so far will be overridden by the predefined personality.

To cancel all changes, the user has to simply close the window by pressing the 'x' button located on the upper right corner.

In the Configuration Dialogue, the personality's settings are distributed in three different tabs. The tab "Basic" (see Figure 9) contains the personality's name, the addresses of the two nodes that constitute the module (device) and a table with all the possible action keys. Only the checked keys will be enabled in the device; that is, the module will only react to a certain action key if the latter has a check next to it in the table. Every key can be enabled and disabled by the user.

By clicking on a device in the table of the main window, this view of the configuration window appears. In the 'Default Configurations' section in the window, the chosen device will be shown with its device name. Since the user can overwrite the name of a personality, it was necessary to show the actuators and the sensors IDs. Furthermore, the current state of the device is prompted so that the user can change the behaviour of a module in that particular state.



Default Configurations: Paul

Current State: Standby

Name: Paul

Actuator Id: 0

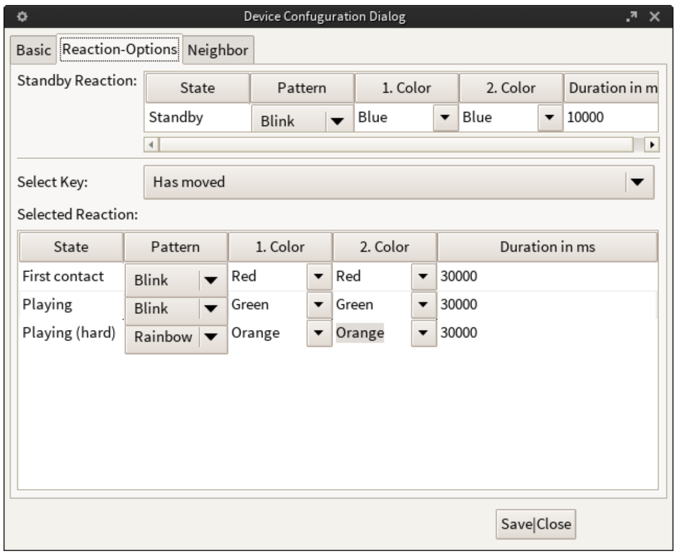
Sensor Id: 0

Action
✓ Rolled Z
✓ Low Battery
✓ Has moved
✓ Found Neighbor
✓ Tipped
✓ Kicked
✓ Rolled X
✓ Rolled Y

Figure 9: Configuration Dialog, 1st Tab

In the second tab of the Configuration Window, the "Reaction Options" can be set. The window is divided in two sections. In the upper one, the behaviour of the device can be set to the Standby (default) state. A personality's behaviour consists of four elements: a pattern (l blinking, fading, rainbow, ...), two colors used by the pattern and finally, the pattern's duration; how long a pattern lasts in time. In the lower part of the Configuration Window, all these elements can be configured by the user.

Finally, the third tab labelled "Neighbor" displays the actions to be performed when the module detects a neighbor (see Figure 11). The table shows the personality of each device that has been detected so far by the server. However, if a module has a preset personality, it will not be displayed in this tab since its reaction towards a neighbour is already



Standby Reaction:

State	Pattern	1. Color	2. Color	Duration in m
Standby	Blink	Blue	Blue	10000

Select Key: Has moved

Selected Reaction:

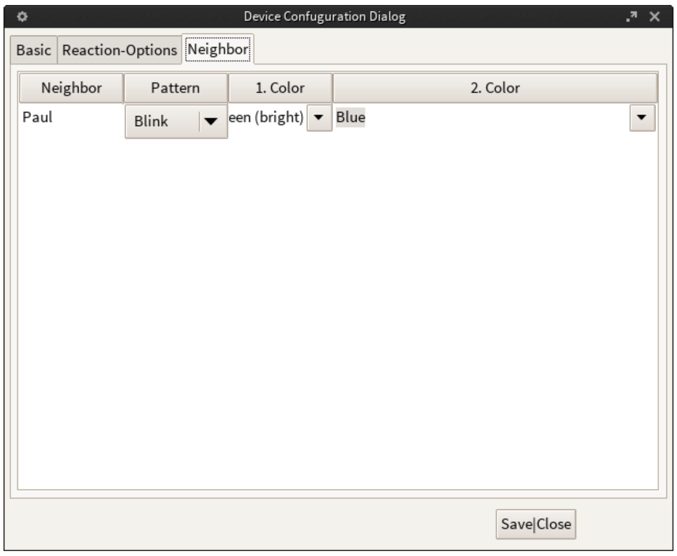
State	Pattern	1. Color	2. Color	Duration in ms
First contact	Blink	Red	Red	30000
Playing	Blink	Green	Green	30000
Playing (hard)	Rainbow	Orange	Orange	30000

Figure 10: Configuration Dialogue, 2nd Tab

defined and is not configurable.

In this tab, the user can set the pattern and colors that the device and its found neighbor will display. For instance, according to the settings in Figure 11, if the current module (that is being configured) finds "Paul", both modules will blink alternating the colors 'green' and 'blue'.

In order to save the changes made to the personality's settings, the user has to press the "save|close" button. Otherwise, if the window is closed without pressing this button before, no changes would be written and the device would keep the settings it had before the configuration process.



Neighbor	Pattern	1. Color	2. Color
Paul	Blink	een (bright)	Blue

Figure 11: Configuration Dialog, 3rd Tab

5.2.3 *Interface Architecture*

The interface was created with the swt widget toolkit from eclipse. To periodically update the device's table, the main window uses a swt timerevent. Every event-based action, like button clicks or selection were also implemented with the built-in event-listener structure of the swt toolkit.

When the Configuration Dialogue window is opened, the module will be passed as a parameter in the Dialogue constructor; the Dialogue constructor will then create the interface and afterwards read the module's settings

Every change will be written to a new device class instance, which shares the same configuration as the selected device, so that if the users closes the configuration dialog via clicking the "save|close" button this instance replaces the previously selected device.

The interface was designed to be highly flexible; if one decides to add new colors or add an extra state to the device and personality structure, the inferface widgets don't need to be changed. They are dynamically created at startup and therefore hold all availably information.

5.3 The Server

To construct the main sever, we chose java[10] as our programming language because of its multi platform abilities and the development suit of eclipse[18] called swt[7] as our window-builder framework. We only used basic java libraries which came by default with the eclipse IDE[20] environment, namely the jdk7-openjdk[3] package.

When the Panstamp server is connected to the Java server, the latter takes the data given by the Panstamp server thanks to the RxTx library [19] and passes the information to the parser for processing.

The server checks with an external timed thread the state of each known device. This thread checks the current state of the device and, if necessary, sets the state back to standby if too much time since the last change has passed. Therefore, the thread compares the timestamp of the last state change with the current time. Furthermore, it checks if the neighbor module is still detected by the device or not.

5.3.1 Panstamp-to-Java server message

All messages sent by the Panstamp server to the Java Server contain four bytes. The message's length was set to a single value to achieve a better communication work-flow and to limit the communication's traffic to a minimum. Table 1 shows the general structure of a message.

Byte 1	Byte 2	Byte 3	Byte 4
Sender's Id	Admin Key	Neighbor's Id| Dummy	Checksum

Table 1: Panstamp-to-Java server message's structure

- **Sender's Id** is the Id of the network node that wants to report an event
- **Admin Key** indicates the event reported by the entity. See table 2 for further information.
- **Neighbor's Id | Dummy** When the Admin Key equals NEARNODEEVENT, this byte contains the Id of the found neighbor node. For other keys this byte is ignored
- **Checksum** This byte is the sum of bytes 1, 2 and 3 (modulo 256)

Table 2 displays all the possible Admin Keys:

Name	Value
Shake Event	31
Kick Event	32
Near Node Event	51
Low battery	52
In Range	53

Table 2: Admin Keys

- **Shake Event** indicates the entity has been shaken slightly so it is swaying
- **Kick Event** means that the entity has been kicked or moved with a strong blow
- **Near Node Event** signifies the entity has detected another entity which is really close to it
- **Low Battery** warns the server that the entity's battery has to be recharged as soon as possible
- **In range** indicates the server that the entity is still in range

5.3.2 The Parser

First, the parser checks whether the length of the package is exactly four bytes and if the last byte, which is supposed to be a checksum summing the byte values of every other bytes in the message, matches the checksum calculated by the server. If everything is correct, the parser then searches for a known device in the modules' list of the server. If it doesn't find a match, the parser will create a new device with default settings.

Once a device was found or created, the parser begins to compare the second byte of the message with the admin keys stored in the server data (see table 2). For example, if the received message contains the number"52" in its second byte, the parser will then recognize this as the "the battery is low" key and will toggle the boolean value battery_low of the device to true.

The keys 'Shake Event', 'Kick Event' and 'Near Node Event' can be ignored by a module if in its settings those events are disabled. In the Configuration Dialog Window, in the 'basic' tab, the user can activate or disable the effect each of those three keys produces in a specific device. Therefore, when a message with one of these keys is received, the parser checks the configured settings of the corresponding device and verifies if the key should be attended to or ignored.

When enabled, the keys 'Shake event' and 'Kick Event' change the module's state to the next stage and led pattern data (pattern and 2 colors) that corresponds to the new module's state will be sent to the server-panStamp using the RxTx-library.

When activated, the 'Near Node Event' key changes the led patterns of the two involved modules (a module and the neighbour it detected). For this, the Java server has to sent two messages (for the two devices) with the same led pattern data to the server-panstamp.

No matter if a correct message leads to a state change or not, the device's timestamp will be updated to the time of the last correct received message. If the module's state did change, the server also stores another timestamp that reflects the time when the last state change happened.

5.3.3 Device and Personality Structure

The device class contains an instance of the personality class and the server class holds a list of all known devices.

Devices can be seen as a representation of the hardware modules with the two panStamps in them. It refers to the hardware and for this reason stores the Id's of the panStamps. Additionally, this class stores a first timestamp of the last received message from a module, a second timestamp of the last module's state change and a third timestamp of the last 'near neighbor' event. These data is accessible through "get" functions as well as mostly modifiable through "set" functions.

The personality class stores the reaction of a particular device depending on the admin keys, neighbors and the state of the device. To implement the set of possible reactions, this class stores a map of reactions (an array of instances of the class 'reaction') with the corresponding activator keys (instances of the enumeration 'keys').

For instance, if the parser receives a message, the reaction map will be searched for the key given by the message. If the key exists in the map, the reaction array will be returned and the personality returns the proper reaction that fits the current state. If the map holds no entry for that key, the reaction map returns a default reaction array.

The possible reactions for each detected neighbor are also stored in a map. This map contains the name of the neighbour' personality and its the corresponding reaction (when this neighbour is found).

Reactions in a device are represented by a class called reaction. This class stores information about colors, patterns and the duration of each reaction (for how long time the reaction will be visible).

To get the server and the project running without setting up every single device by hand, a enumeration for pre defined personalities was created. One can select between each of those personalities. Once a pre defined personality is selected, the previous personality of the current device will be overwritten. It is also possible to save the current devices to a file so that they can be loaded again after restarting the server.

5.3.4 Key Structure

During the development of the system, we decided to use single byte keys to represent commands, messages, patterns and colors inside the Java server. To store these keys, several enumerations were created to separate related keys with their name and value depending on their purpose. We divided the keys in enumerations:

- system
  - stores basic keys for transmitting status information (e.g. "low battery")
  - enumeration: system_keys
- color
  - stores byte values for every color hard coded to the actuator panStamp of the hardware device
  - enumeration: color
- pattern
  - stores byte values for every pattern hard coded to the actuator panStamp of the hardware device
  - enumeration: pattern
- event key
  - stores byte values for every possible action happening to the hardware device
  - enumeration : keys

Every class or function that refers to those values calls only the enumeration item by its name. With this implementation, it is easier to change single values and it is quicker to get an overview.

6. DRIVER

The Panstamps' drivers were developed in C++. Each member in the network, namely serverNode, actuatorNode and sensorNode, has its own driver; nevertheless, all of them are based on the class "ccNode" as shown by the UML graph in Figure 13. In other other words, ccServer, ccActuatorNode and ccSensorNode are all subclasses of the superclass "ccNode".

All the nodes classes' names start with "cc" to denote that their communication depends completely on the class CC1101, which manages the transceiver integrated in each Panstamp.

6.1 ccpacket class

The class ccpacket contains the backbone for all the messages (packages) shared by the network's nodes. As depicted in Figure 13, this class' attributes are:

- length, which determines the size of each package
- data, an array that holds the packet information
- crc_ok. This boolean value set by the transceiver, upon receiving a packet, indicates the packet's data integrity.

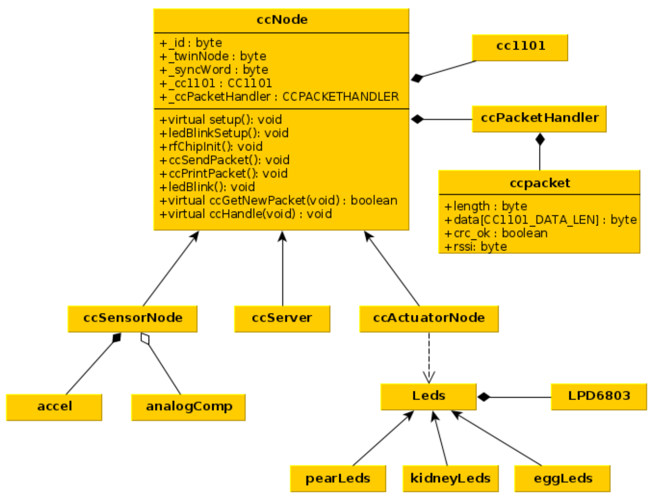


Figure 13: Driver's short UML

- rssi. This value is set by the CC1101 that receives the packet and indicates the received signal strength.

In our implementation, the packet's data length was set to 5 bytes. There are two types of packets:

- Notification packets
- Command packets

6.1.1 Notification packets

Notification packets can be sent by all nodes of the network; they inform other nodes about current events. For example, when a node is moved or kicked, it broadcasts a notification packet to all nodes to inform about this occurrence.

Table 3 shows Notification packet's structure:

Name	Value
Receiver Id	data[0]
Sender Id	data[1]
Admin Key	data[2]
Near Node Id or Dummy	data[3]
Dummy	data[4]

Table 3: Notification packet

Depending on the value of "Receiver Id", a message is received by all nodes or just by a single one. To broadcast, "Receiver Id" has to be set to "0" (zero). In other words, a packet is broadcasted when packet.data[0] is set to "0" [13]. When the packet has to be sent to just a single node, "Receiver id" holds the receiver node's id.

The "Sender Id" indicates which node created and sent the packet. This data is important for interpreting any packet's meaning.

"Admin Key" points out an specific event perceived by a node in the network. When a notification packet is received by the server, this node creates a new Panstamp-to-Java server message (see section 5.3.1) to inform the Java server about the new event. Notification packets share the same keys as the Panstamp-to-Java server messages; to see all possible values that "Admin Key" can take, please refer to Table 2.

When a module is moved or kicked, its sensor node broadcasts a notification packet (with "Admin Key" *shake_event* or *kick_event*) that is received by all the sensor nodes and by the server node. Each sensor node compares the rssi of the received notification packet to a specific threshold to determine if the node that broadcasted the notification packet is near. When the rssi is strong enough, the sensor node sends a notification packet to the server node with "Admin Key" equal to *Near_Node_Event*. In this Notification packet, 'data[3]' contains the id of the sensor node that was detected to be near. For the rest of the Notification packets, 'data[3]' doesn't contain any real data (is a dummy value).

To keep the same length for all packets, the Notification packets has a dummy byte in his data array ("data[4]"). Nevertheless, this data byte contains real data in the Command packets.

6.1.2 Command packets

Command packets are sent only by the server node to the actuator nodes.

Table 4 shows the Command packet structure:

Name	Value
Receiver Id	data[0]
Sender Id	data[1]
Metakey	data[2]
Color1	data[3]
Color2	data[4]

Table 4: Command packet

The two first fields, Receiver Id and Sender Id, are the same as the ones contained in the Notification packet section (see section 6.1.1). In a Command packet, the Sender Id byte will be always "1", the server Id, because only the server send Commands Packets.

"Metakey" indicates to the actuator node what pattern has to be displayed by the led strand. Table 5 contains the patterns implemented in the system.

Name	Value
Blink	0
Fade	1
Rainbow	2
Leds On	3
Leds Off	4
Onestripe	5
Stripes	6

Table 5: Meta Keys

Most of the patterns displayed by the led strands have one or two colors. Fields "Color1" and "Color2" contain keys that represent colors. The color pallet implemented in this system has 13 key colors but this pallet can be easily extended.

6.2 ccPacketHandler

The ccPacketHandler class is in charge of managing the creation of packets and the access of data in received packets. For these tasks, the class contains an instance of the ccPacket class (see Section 6.1).

6.3 ccNode

As shown Figure 13, the ccNode is an abstract class that contains the basic methods and attributes shared by all the different nodes' classes.

6.3.1 ccNode methods

The most important methods in this superclass are:

- virtual setup() sets the node's attributes depending on its type. See section 6.3.2
- rfChipSetup() initializes and configures the CC1101 chip
- ccSendPacket() sends packets to other nodes in the network
- ccGetNewPacket() receives packets from other nodes. Its implementation is different in each type of node.
- ccHandle() interprets the data contained in a packet; therefore, its implementation varies among the nodes' types.

6.3.2 ccNode attributes

Every node in the network has the following attributes:

- _id stores the node's id
- _twinNode has the id of the other node contained in the same module. For instance, in a sensor node this attribute is the id of the actuator node contained in the same module.
- _syncWord identifies the network. All nodes in our network share the same syncWord.
- _cc1101 is an instance of the cc1101 class that controls the panstamp's transceiver
- _ccPacketHandler is an instance of the ccPacketHandler class (see section 6.2).

6.4 ccSensorNode

The ccSensorNode class creates and sends notification packets based on the information provided by the physical sensors, namely the accelerometer and the voltage monitoring circuit. Additionally, it creates and sends a notification packet when another module is really close. We consider that two modules are closed to each other when the distance between them is about 30 cm or less.

To perform its tasks, the ccSensorNode class uses the accel and analogComp classes as shown in Figure 13.

6.4.1 Accel class

The accel class reads the signals given by the accelerometer and determines, according to specific thresholds, if a module was shaken or kicked. Since each module has a different mass, the acceleration produced by the same force varies among the modules. Therefore, when an instance of the accel class is created, two thresholds are required: shakenThreshold and kickedThreshold. In our system, these thresholds were estimated by sampling and averaging the acceleration data in each different module (pear, kidney and egg).

In section 7.1.1, we indicate that the accelerometer ADXL355 mounted in the board GY-61 has a bandwith of 50 Hz; i.e., its output signal has a frequency of 50 Hz. According to Nyquist–Shannon sampling theorem, a band-limited signal can be reconstructed from a countable sequence of samples if the signal's bandwidth is no greater than half the sampling rate. Therefore, the accel class samples the accelerometer's signal at a frequency of 200 Hz (1 sample each 5 milliseconds). As a result, the analog-to-digital sampling frequency is 4 times the accelerometer's signal bandwidth, which is greater than two times the accelerometer's bandwidth.

6.4.2 AnalogComp class

With the AnalogComp class, the panstamps' Analog Comparator module can be activated and configured to enable interruptions. As explained in section 7.1.2, in our implementation, the Analog Comparator triggers an interruption in the sensor node when the voltage on the pin AIN0 (digital pin D6) decreases below the voltage in pin AIN1 (digital pin D7).

6.5 ccServer class

The ccServer class' main purpose is to enable the communication between the Java Server and the other network nodes.

First, this class receives Notification packets sent by the sensor nodes and transfers the data contained in these packets to the Java Server. Messages transferred from the panstamp server to the Java server are explained in section 5.3.1.

Second, the ccServer class receives commands from the Java Server and creates Command packets accordingly to the corresponding actuator nodes.

6.6 ccActuatorNode class

The ccActuatorNode class is in charge of receiving Command packets and of transferring the pattern data to the Leds class.

6.7 Leds class

The Leds class controls the led strand connected to the actuator node. It receives pattern data from the ccActuatorNode class and reproduces the expected led pattern in the module. Since each module type has a different number and spatial distribution of leds, a subclass of the Led class was created for each type of module, namely pearLeds, kidneyLeds and eggLeds.

6.7.1 *LPD6803 class*

This class, published originally by ladyada in github [8], provides the methods for settting-up and for controlling each led in the led strands we used in our system.

7. HARDWARE

7.1 Sensor Node

The sensor node is constituted by the panstamp, an accelerometer and a voltage divisor circuit.

7.1.1 *Accelerometer*

The ADXL335 is a small low power 3-axis accelerometer that measures acceleration with a range of at least +/- 3g; in other words, +/- 3g is the maximum amplitude that this accelerometer can detect before distorting or clipping its output signal. 1g is the acceleration due to the earth's gravity (9.8 m/sec2).

According to the ADXL335 datasheet [2], the capacitors Cx, Cy and Cz connected to the accelerometer pins Xout, Yout and Zout determine the accelerometer's bandwidth. With these capacitors, a low pass-filter for anti-aliasing and noise reduction is implemented. Aliasing denotes the distortion and artifacts that can appear when a signal is reconstructed from samples of the original signal. In the board GY-61 (see Figure 14), Cx, Cy and Cz have a capacitance of 0.1uF that yields a bandwidth of 50 Hz [2].

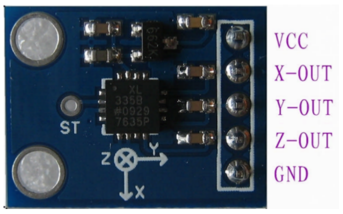


Figure 14: ADXL335 accelerometer in board GY-61

7.1.2 *Voltage monitoring circuit*

To detect when the Li-Po battery has low voltage, a voltage monitoring circuit was implemented with the Panstamp's ATMEGA 328 analog comparator module [5] and a voltage divisor circuit (see Figure 15). According to the ATMEGA 328 data-sheet [5], the Analog Comparator compares the input values on the positive pin AIN0 and negative pin AIN1.

In our implementation, the Analog Comparator triggers an interruption in the sensor node when the voltage on the pin AIN0 decreases below the voltage in pin AIN1. With the configuration shown in Figure 15, the voltage in AIN1 is always 2.4 V. When the Li-po battery voltage falls below 9.5 Volts, the voltage in AIN0 (which is the same voltage in resistor R2) is less than the voltage in AIN1 and the Analog Compator activates an interruption.

To minimize power dissipation in the voltage monitoring circuit, the possible maximum current in brach R1-R2 and in branch R3-R4 was fixed to 0.1mA. With this value and the maximum voltage given by the Li-po battery (12.5 V) the values of R1, R2, R3 and R4 were calculated.

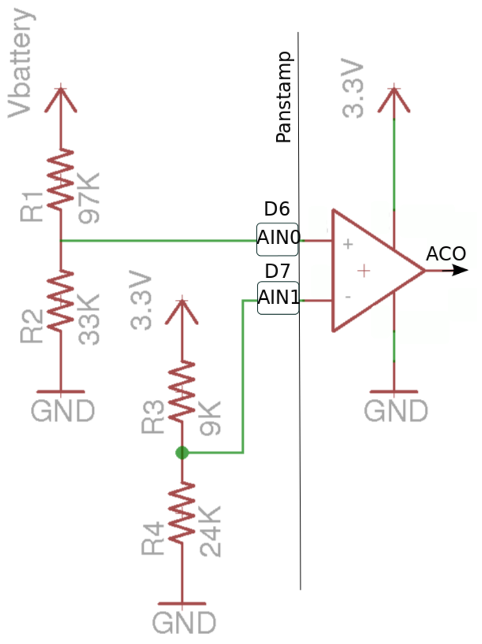


Figure 15: Voltage monitoring circuit

7.2 Actuator Node

The actuator node is constituted by a panstamp and a led strand, whose length varies among the different modules.

7.2.1 *Led strand*

In the led strand, each pixel contains a small microchip called LPD6803 and both led and chip are encapsulated within a silicone enclosure . Thanks to this microchip, each pixel in the strand can be controlled individually. It is recommended to use a supply voltage of 5 volts for the led strand [1]. As shown in Figure 16, the led strand has 4 cables: the red and white ones are for the power supply (red is positive and white has to be connected to ground), the blue one is the data cable and finally the green one is the clock cable. In our system, blue is connected to the panstamp's digital pin 5 and the green cable is connected to the panstamp's digital pin 3.

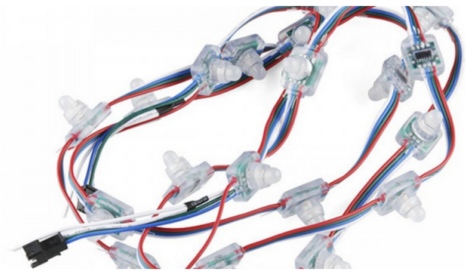


Figure 16: Led strand

7.3 Server Node

The sensor node is constituted by a panstamp and a panStick (See Figure 17). PanStick is a USB mother board that enables both the programming of panStamps and the communication between a panstamp and a computer via USB.

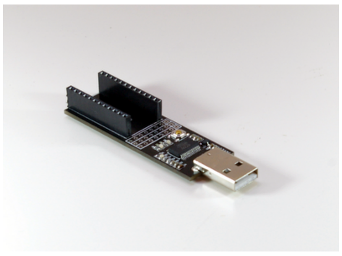


Figure 17: Panstick

Additionally, the panStick includes a reset button that lets restart an application without having to unplug the board.

8. MINIPROJECT: THE WHEEL

Before the main project's goal was defined, we studied and made experiments with the different sensors we had at hand, such as the hall sensor. We took a plastic wheel and attached small magnets on the wheel's spokes. Then, we made two small holes in the wheel's support and in each one we put a hall sensor. A small circuit with a panstamp was build to analyse the hall sensors' output signals. With the hall sensors signal pattern, we could determine the direction of movement and also the number of turns made in a period of time by the wheel. Figure 18 shows the wheel along with the panstamp.

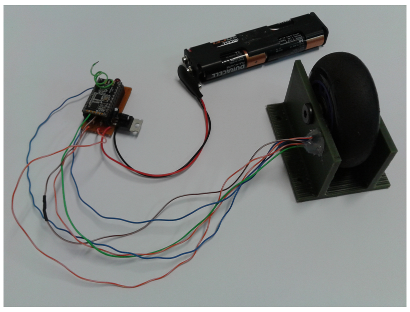


Figure 18: The wheel and panstamp

8.1 The hall sensor

The sensor TLI4906 is a Hall Effect Switch, whose output depends on the magnetic field sensed by the chip (see). According to the datasheet, a magnetic south pole with a field strength above Bop (typical value 10 mTeslas) turns the output on, whereas a magnetic north pole exceeding Brp (typical value 8.5 mTeslas) turns it off. Figure 19, taken from the datasheet, shows the relationship between the output voltage (named Vq) and the magnetic field sensed by the hall effect ship.

After experimenting with the sensor, we realized that the absence of a magnet in front of the ship also turns the output off. With this property, we could distinguish two possible states: magnet in front of the chip and no magnet in front of it. Since we were using to hall effect sensors, we could have 4 different states, as shown in Table 6.

When the wheel rotates, the position of the magnets changes and this alters the combination of outputs of the hall effect

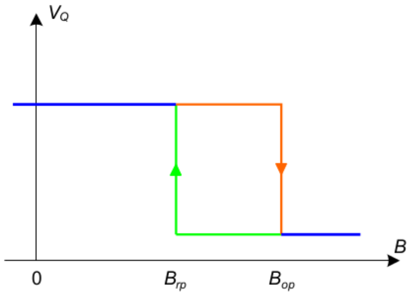


Figure 19: The hall sensor's output

Sensor 1	Sensor 2	Vsensor1	Vsensor2
No magnet	No magnet	0	0
No magnet	Magnet	0	1
Magnet	No magnet	1	0
Magnet	Magnet	1	1

Table 6: Possible output voltages combinations with the two hall effect sensors

sensors. The sequence of different states yields a pattern that can be used to determine the wheel's rotation direction (clockwise or counterclockwise) In the next section, we describe how the hall effect output signals were processed by the panstamp.

8.2 Processing the hall effect sensors' signals

Figure 20 shows a simple diagram of the circuit built to determine the wheels direction of rotation.

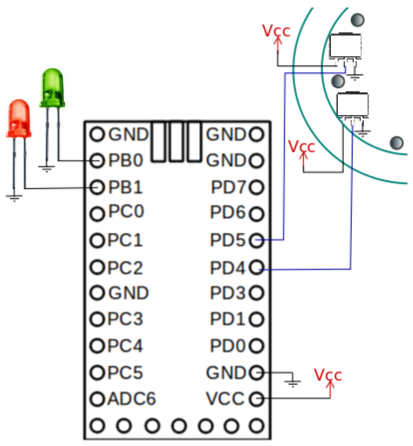


Figure 20: Wheel circuit

The output pin of each hall effect sensor was connected to a panstamp's digital output pin, namely PD4 and PD5.We configured an interruption, in the panstamp code, that is triggered with each change (rise and fall) in the voltage sensed by pin PD4. In interruption routine, the voltages in PD4 and PD5 were measured and a flag was raised. These voltage measurements were interpreted as binary inputs ('0' or '1') and constituted the trace that reflected the direction of rotation. The location of each magnet in the wheel's spokes was carefully selected to produce clear differences between the clockwise and the counterclockwise trace. The raised flag, mentioned before in the interruption routine, indicated if there was a change or not in the state; that is, if the wheel was rotating or not.

Without the library 'PinChangeInt.h', the only digital pins that can trigger this interruption are digital pins PD2 and PD3. Since the use of PD2 cannot be customized in the Panstamp, we had available just PD3. At the beginning, we thought we needed two interruptions (one triggered by PD4 and another by PD5) but Later, we realized that one interruption sufficed to do the analysis. In other words, we could have used pins PD3 and PD4 and configured interruptions with PD3 without the help of the above mentioned library.

As explained in the arduino official website [4], it is useful to steer a digital input pin to a known state if no input is present. Writing a HIGH value with digitalWrite() in a digital pin, if if the pin is configured as an INPUT, will enable an internal 20K pullup resistor (to Vcc). On the other hand, writing LOW will disable the pullup. In our implementation, adding the internal pullup was fundamental to obtain the right readings of the hall effect sensor's output signals.

Each time the panstamp detected movement in the wheel, it built a packet and sent it wirelessly to another panstamp that lit up a led strand. Two different packets were sent according to the direction of rotation and the leds in the led strand were lit in a way that reflected the two different directions. For debugging purposes, a led was lit when a certain direction of rotation was detected: the green led indicated clockwise rotation whereas the red one counter-clockwise rotation.

This mini-project is a simple example of how the hall effect sensors, along with a panstamp, can be used to produce interactive systems.

9. CONCLUSION

The development of our system required a balanced combination of creativity and technology. First, we analysed already existing projects for public spaces and then some systems where the foot played the main goal. Then, after defining our own concept, we started to explore different possibilities for implementing each necessary component in the concept and chose the options that offered more flexibility and overall utility.

References

[1] ADAFRUIT. *LPD6803 datasheet.* `http://www.adafruit.com/datasheets/LPD6803.pdf`.

[2] ANALOG DEVICES. *ADXL335 datasheet*, 2009. `https://www.sparkfun.com/datasheets/Components/SMD/adxl335.pdf`.

[3] ARCH LINUX. *JRE7-Openjdk Arch Linux official repository entry.* `https://www.archlinux.org/packages/?name=jdk7-openjdk`.

[4] ARDUINO. *Arduino official website*, April 27, 2012 11:17. `http://arduino.cc/en/Tutorial/DigitalPins`.

[5] ATMEL. *ATMEGA328 webpage*, 2013. `http://www.atmel.com/devices/atmega328.aspx`.

[6] DIGI INTERNATIONAL INC. *XBee Product Page.* `http://www.digi.com/products/xbee/`.

[7] ECLIPSE FOUNDATION. *SWT Homepage*, 2013. `http://www.eclipse.org/swt/`.

[8] LADYADA. *Lady Ada LPD6803 driver*, 2011. `https://github.com/adafruit/LPD6803-RGB-Pixels`.

[9] LADYADA. *Lady Ada Arduino Tutorial*, April 27, 2012 11:17. `http://www.ladyada.net/learn/arduino/`.

[10] ORACLE. *Java Homepage.* `https://www.java.com/en/`.

[11] PAELKE, V., REIMANN, C., AND STICHLING, D. *Foot-based mobile interaction with games.* New York, NY, USA, 2004.

[12] PAKKANEN, T., AND RAISAMO, R. *Appropriateness of foot interaction for non-accurate spatial tasks.* New York, NY, USA, 2004.

[13] PANSTAMP. *CC1101 Library.* `http://code.google.com/p/panstamp/wiki/LowLevelLibrary`.

[14] PANSTAMP. *Panstamp official website*, April 27, 2012 11:17. `http://www.panstamp.com/home`.

[15] SEITINGER, S., PERRY, D. S., AND MITCHELL, W. J. *Urban pixels: painting the city with light.* New York, NY, USA, 2009.

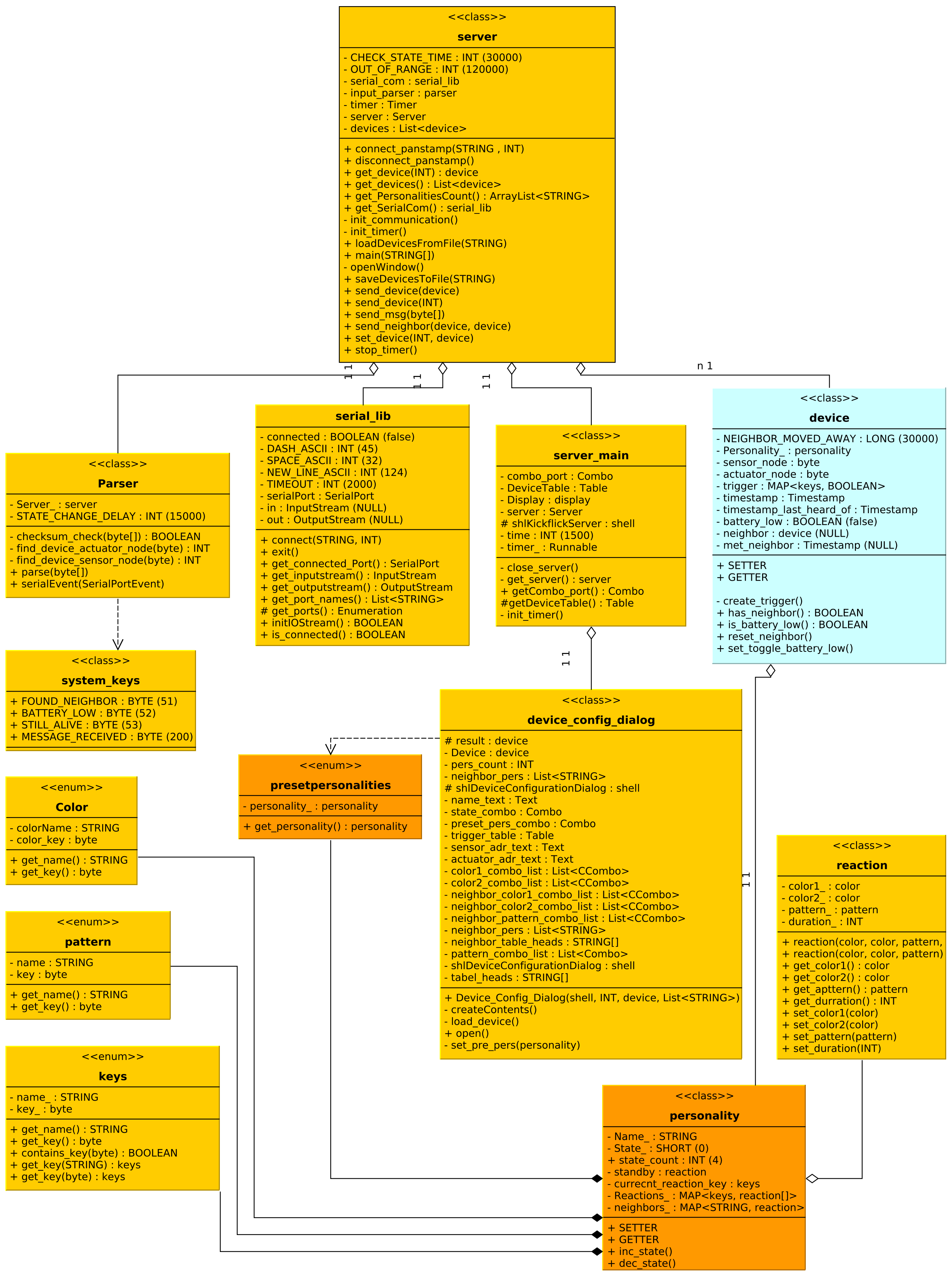
[16] SEITINGER, S., TAUB, D. M., AND TAYLOR, A. S. *Light bodies: exploring interactions with responsive lights.* New York, NY, USA, 2010.

[17] TEXAS INSTRUMENTS. *CC1101 datasheet*, 2013. `http://www.ti.com/lit/ds/symlink/cc1101.pdf`.

[18] THE ECLIPSE FOUNDATION. *Eclipse Homepage*, 2013. `http://eclipse.org//`.

[19] TRENT JARVI. *RxTx Library Wiki.* `http://rxtx.qbang.org/wiki/index.php/Main_Page`.

[20] WIKIMEDIA FOUNDATION. *IDE Wikipedia Entry.* `https://en.wikipedia.org/wiki/Integrated_Development_Environment`.



<<class>>
server
- CHECK_STATE_TIME : INT (30000)
- OUT_OF_RANGE : INT (120000)
- serial_com : serial_lib
- input_parser : parser
- timer : Timer
- server : Server
- devices : List<device>
+ connect_panstamp(STRING , INT)
+ disconnect_panstamp()
+ get_device(INT) : device
+ get_devices() : List<device>
+ get_PersonalitiesCount() : ArrayList<STRING>
+ get_SerialCom() : serial_lib
- init_communication()
- init_timer()
+ loadDevicesFromFile(STRING)
+ main(STRING[])
- openWindow()
+ saveDevicesToFile(STRING)
+ send_device(device)
+ send_device(INT)
+ send_msg(byte[])
+ send_neighbor(device, device)
+ set_device(INT, device)
+ stop_timer()
1 1
1 1
1 1
n 1
serial_lib
- connected : BOOLEAN (false)
- DASH_ASCII : INT (45)
- SPACE_ASCII : INT (32)
- NEW_LINE_ASCII : INT (124)
- TIMEOUT : INT (2000)
- serialPort : SerialPort
- in : InputStream (NULL)
- out : OutputStream (NULL)
+ connect(STRING, INT)
+ exit()
+ get_connected_Port() : SerialPort
+ get_inputstream() : InputStream
+ get_outputstream() : OutputStream
+ get_port_names() : List<STRING>
# get_ports() : Enumeration
+ initIOStream() : BOOLEAN
+ is_connected() : BOOLEAN
<<class>>
server_main
- combo_port : Combo
- DeviceTable : Table
- Display : display
- server : Server
# shlKickflickServer : shell
- time : INT (1500)
- timer_ : Runnable
- close_server()
- get_server() : server
+ getCombo_port() : Combo
#getDeviceTable() : Table
- init_timer()
<<class>>
device
- NEIGHBOR_MOVED_AWAY : LONG (30000)
- Personality_ : personality
- sensor_node : byte
- actuator_node : byte
- trigger : MAP<keys, BOOLEAN>
- timestamp : Timestamp
- timestamp_last_heard_of : Timestamp
- battery_low : BOOLEAN (false)
- neighbor : device (NULL)
- met_neighbor : Timestamp (NULL)
+ SETTER
+ GETTER
- create_trigger()
+ has_neighbor() : BOOLEAN
+ is_battery_low() : BOOLEAN
+ reset_neighbor()
+ set_toggle_battery_low()
<<class>>
Parser
- Server_ : server
- STATE_CHANGE_DELAY : INT (15000)
- checksum_check(byte[]) : BOOLEAN
- find_device_actuator_node(byte) : INT
- find_device_sensor_node(byte) : INT
+ parse(byte[])
+ serialEvent(SerialPortEvent)
<<class>>
system_keys
+ FOUND_NEIGHBOR : BYTE (51)
+ BATTERY_LOW : BYTE (52)
+ STILL_ALIVE : BYTE (53)
+ MESSAGE_RECEIVED : BYTE (200)
1 1
<<class>>
device_config_dialog
# result : device
- Device : device
- pers_count : INT
- neighbor_pers : List<STRING>
# shlDeviceConfigurationDialog : shell
- name_text : Text
- state_combo : Combo
- preset_pers_combo : Combo
- trigger_table : Table
- sensor_adr_text : Text
- actuator_adr_text : Text
- color1_combo_list : List<CCombo>
- color2_combo_list : List<CCombo>
- neighbor_color1_combo_list : List<CCombo>
- neighbor_color2_combo_list : List<CCombo>
- neighbor_pattern_combo_list : List<CCombo>
- neighbor_pers : List<STRING>
- neighbor_table_heads : STRING[]
- pattern_combo_list : List<Combo>
- shlDeviceConfigurationDialog : shell
- tabel_heads : STRING[]
+ Device_Config_Dialog(shell, INT, device, List<STRING>)
- createContents()
- load_device()
+ open()
- set_pre_pers(personality)
1 1
<<enum>>
presetpersonalities
- personality_ : personality
+ get_personality() : personality
<<enum>>
Color
- colorName : STRING
- color_key : byte
+ get_name() : STRING
+ get_key() : byte
<<enum>>
pattern
- name : STRING
- key : byte
+ get_name() : STRING
+ get_key() : byte
<<enum>>
keys
- name_ : STRING
- key_ : byte
+ get_name() : STRING
+ get_key() : byte
+ contains_key(byte) : BOOLEAN
+ get_key(STRING) : keys
+ get_key(byte) : keys
<<class>>
reaction
- color1_ : color
- color2_ : color
- pattern_ : pattern
- duration_ : INT
+ reaction(color, color, pattern,
+ reaction(color, color, pattern)
+ get_color1() : color
+ get_color2() : color
+ get_aptern() : pattern
+ get_durration() : INT
+ set_color1(color)
+ set_color2(color)
+ set_pattern(pattern)
+ set_duration(INT)
<<class>>
personality
- Name_ : STRING
- State_ : SHORT (0)
+ state_count : INT (4)
- standby : reaction
- currecnt_reaction_key : keys
- Reactions_ : MAP<keys, reaction[]>
- neighbors_ : MAP<STRING, reaction>
+ SETTER
+ GETTER
+ inc_state()
+ dec_state()

Figure 12: UML Diagram of the Server Structure