A Coordinated Matrix of RFID Readers as Interactions Input

Maxime Louvel, François Pacull
CEA-LETI MINATEC Campus, Grenoble, France
Email: maxime.louvel@cea.fr ; francois.pacull@cea.fr

***Abstract*—The paper presents a framework to develop applications on a very innovative hardware associating hundreds of rfid readers and a high resolution display within a table. The framework is built on top of a rule-based coordination middleware, which provides mechanisms to handle combinations of events, generated by the rfid readers. The framework offers the basic blocks to fully support the hardware. The paper demonstrates the interest and the possibilities of the framework through simple examples and a more complex scenario. Both illustrate how easy it is to build any kind of interactions with the proposed framework.**

Keywords*—*Coordination Middleware; RFID; Data aggregation.

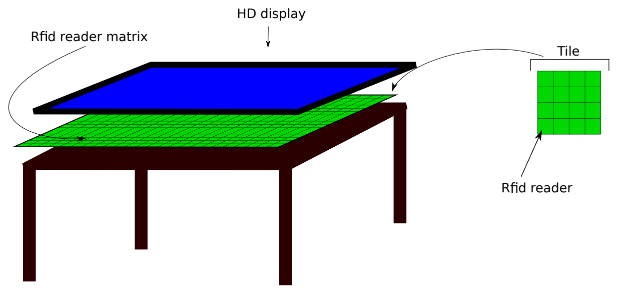

Fig. 1. Description of the table

I. Introduction

Sensor networks are continuously growing and bringing new designs and usages. The increasing number of devices implied at the same time and the increasingly complex interactions required by the usages does not ease the task of the application programmers. There is a need of a middleware layer, offering as basic bricks high level mechanisms, in order to move most of the complexity from the application to the middleware. This paper illustrates this with an innovative smart table hosting a high resolution display and a matrix of several hundreds of rfid readers. The usage of this table is multiple when it is question of interaction, mediation and collaboration between several users.

The paper is organised as follows. Section II describes the hardware used, an innovative table that allows to detect the identity and the position of rfid tagged objects put on the table and to display arbitrary picture on the HD screen. Section III presents the rule based middleware and the framework built on top, which offers to the application designer the basic interaction involving objects equipped with rfid tags and the 2D graphical engine playing the role of interactive and dynamic tablecloth. Section IV illustrates simple usages of the framework and a more advance scenario. Finally, section V concludes the paper.

II. Hardware

To illustrate the capability of our middleware to manage complex events detection, we describe our experiment with an original hardware. This hardware combines a rfid based location system and a HD screen that can be used as classical display.

Fig. 1 depicts the table, composed of two layers. The first layer is a 42” screen able to display with a resolution of HD 1080p. This screen is seen as a classical LCD display and can thus be connected to a computer or a raspberry pi board in this paper. Under this display layer, there is a set of rfid readers organised as a matrix of 6 x 4 tiles, with each tile containing itself a matrix 4 x 4 rfid readers. As a result there are 24 x 16 (384) rfid readers distributed in the table.

This table works with classical rfid tags that can be attached to any physical object. The raw information received is for each rfid reader the set of detected tags. This information is collected via Ethernet. Each tile has its own IP address and gives information for the 16 rfid readers constituting the tile.

There are two interesting functioning modes of this table. In the first mode (push), the tiles are autonomous and send automatically information each time a rfid is seen. In the second mode (pull), each tile can be interrogated in order to have the information.

With this hardware, the applicative fields are quite infinite provided that the middleware offers the required abstraction layer and a powerful mechanism to define the coordination scheme we want to put in place.

III. Software

The presented hardware allows a lot of interaction through objects. It needs a high level middleware able to quickly react to the context defined by the set of objects present on the table at the same time. Applications for this hardware typically combine rfid tag location, co-location (several tags), proximity, distance, sequence of tags put on the table. Moreover it is possible to use other interfaces connected to the system (e.g., 3d mouse, cameras). This section firstly introduces the middleware we use. For a more detailed description of this middleware, the reader may refer to [1]. Then the section presents the framework we developed on top to ease the creation of applications using the table.

A. Coordination Middleware

This middleware is an evolution of earlier middlewares [2], [3] specifically designed for lightweight systems. It provides a uniform abstraction layer that eases the integration and coordination of the different components (software and hardware). It relies on the *Associative memory* paradigm implemented in our case as a distributed set of bags containing resources (tuples). Following Linda [4] approach the bags are accessed through the three following operations:

- `rd()` which takes as parameter a partially instantiated tuple and returns from the bag a fully instantiated tuple whose fields match to the input pattern;
- `put()` which takes as parameter a fully instantiated tuple and insert it in the bag;
- `get()` which takes as parameter a fully instantiated tuple, verifies its presence in the bag and consumes it in an atomic way.

For a matrix of rfid readers like the one described above, bags `RawInformation` and `Position` may contain raw data such as `(tagid, tileid, readerid)` or more refined data as `(tagid, posX, posY)`. Depending on the usage (calibration or real application) they both have an interest. Once the location is computed according to the raw data meta-data may be considered from the association between `(physicalTagId, tagId)` or `(tagId, objectId).`

For actuators, the `put()` operation is used to insert tuples under the form `(actuatorid, function, parameter1, parameter2)`. Once inserted in the bag, it actually triggers the correct action on the physical actuator with the appropriate parameters. The same `put()` operation inserts tuples into bags configuring the readers operating mode or other configuration parameters. Finally, some bags are used to control the videos which are displayed on the table screen.

In addition, bags can be grouped inside objects. For instance an object can model the display on the table and another may handle all the rfid readers.

The operations `rd(), get()` and `put()` are used in the Production rules [5] to express the way these resources are used in the classical *pre-condition* and *performance* phases.

Precondition phase: It relies on a sequence of rd() operations to find and detect the presence of resources in several bags. This can be sensed values, result of service calls or states stored in tuplespaces or databases.

The particularity of the precondition phase is that:

- the result of a `rd()` operation can be used to define some fields of the subsequent `rd()` operation;
- a `rd()` is blocked until a resource corresponding to the pattern is available.

Performance phase: It combines the operations `rd()`, `get()` and `put()` to respectively verify that some resources found in the precondition phase are still present, consume some resources and insert new resources. In this phase, the operations are embedded in *distributed transactions*[6]. This particularity ensures several properties that go beyond traditional production rules. In particular it ensures that:

- the conditions responsible of firing the rule (precondition) are still valid in the performance phase;
- the different involved bags are actually all accessible.

These properties are very important since they allow to verify that a set of objects are actually present "at the same time" on the table.

B. Framework

The proposed framework is composed of three objects: *`Rfid`*, *`Display`* and *`2D_Engine`*.

1) Object `Rfid`: This object models the Rfid readers matrix. It contains the following bags:

- `Position(tagId, posX, posY)`: contains the position of the tag (0,0 defines the top left position);
- `LogicalTag(physicalTagId, tagId)`: stores the association of a physical tagId with a more meaningful logical id e.g. ("030209348393", "video1");
- `TagStatus(tagId, status)`: contains the status of a tag: `"in"` if detected by a rfid reader or `"out"` if not seen for a given delay;
- `Mapping(tagId, objectId)`: keeps the association physical object and rfid tag that is attached to it (e.g., an hourglass used to symbolise a timer);
- `Type(tagId, type)`: maintains association of a tagId with a type of tagged object (e.g., physical object, video, action card, badge);
- `Area(areaId, areaDefinition)`: contains areas on the table defined as a set of points defining a polygon;
- `PositionArea(tagId, areaId)`: contains the tagId contained in a given area.

The detection of the tags placed on the table is done by a driver which handles the events sent by the different rfid readers (used in push mode). This information is decoded and the different bags are filled with the corresponding resources. When a tag is detected, the driver computes its position on the table (X,Y) and adds the resource `(tagId, posX, posY)` in the bag `Position`. If several readers detect the same `tagId`, the barycentre is computed. The computation uses as weight the signal strengths of the readers seeing a tag in order to improve the precision of the location. As a rfid reader continuously sends the tag information, a filtering is applied to avoid inserting new resources when it is not necessary. Then, a resource is inserted only when a significant change in the location is effective. In addition, the `status` of the tag, `"in"` if the tag is still on the table or `"out"` if it has been removed (i.e., not be seen for a given delay), is inserted as a resource `(tagId, status)` in the bag `TagStatus` each time the `status` changes.

The bags `Type`, `Area` or `LogicalTag` are configuration bags and their usage is described here after.

a) Introduction to rules: The described middleware allows to express with its rule based language actions to be performed (performance phase) when some conditions (precondition phase) are verified. The actions performed are embedded in transactions enclosed in {}. As `rd()` actions may be included in these transactions, it is possible to ensure that resources found in the precondition are still valid in the performance.

```
1 ["Rfid","TagStatus"].rd(tagId, "in") &
2 # other preconditions
3 ::
4 {
5  ["Rfid","TagStatus"].rd(tagId, "in");
6  # other actions
7 }.
```

Fig. 2. Ensure tag is still there at performance phase

Fig. 2 presents an example of rule, where the precondition and performance part are respectively before and after the `"::"`. To simplify the example, we only show a single operation in the precondition and performance phase but both may contain several additional tokens.

The first token (line 1) reads in the bag `TagStatus` of the object `Rfid` all the tags with status "in". This allows to detect new tags placed on the table and then to manage the corresponding scenario. Line 5 guaranties that the tagId is still on the table when the performance phase executes. Since actions in the performance are embedded in transactions the other actions can only be done if the tag is still there. Note that this approach simplifies a lot the management of events:

- events are detected in preconditions;
- when performances are executed, guaranteeing that the condition related to the event is still valid only requires to add a `rd()` in the performance part.

b) Initialisation rules: Fig. 3 presents an initialisation rule. No precondition is defined, this rule is always executed and only once at the application launch time.

```
::
{
 ["Rfid","LogicalTag"].put("9e7f9cce9","tag_video_table");
 ["Rfid","Area"].put("zoneA","0,0;0,54,12,54;12,66;66,0");
 ["Rfid","Type"].put("t_video_presentation_table","video");
}.
```

Fig. 3. Initialisation rule

Here we initialise the bags `LogicalTag`, `Area` and `Type`.

In the first bag, we associate the physical tagid `"94e7f89cce9"` to the more user friendly logical tag `"tag_video_table"`. This allows to manipulate in the rules an id that is human readable. In addition, several physical tags can be associated to the same logical tag for backup reason or to offer to several people the possibility to trigger the same action with different objects or cards.

In the second bag, we define a `"zoneA"` as a list of points defining a polygon. This is taken into account by the driver to populate the `PositionArea` bag.

In the third bag, we associate a `type` to a tag. The `type` allows to define a specific context around this tag to verify that it is correctly used. For instance, a tag associated to a voting card cannot be placed everywhere on the table but in a given area. Another usage is to give information to the driver about the sampling frequency for a given tag or if the change in the location is large enough to be reported or not.

c) Defining action area: To better organize the table, area (i.e., zone of the table) can be used. An area is defined by adding a resource `(areaId, areaDefinition)` in the bag `Area`. The `areaDefinition` is a set of points defining a polygon. When the rfid driver detects a new position for a tag, at the same time it inserts the corresponding resource in the bag `Position`, it scans all the defined areas and add the resources `(tagId, areaId)` in the bag `Area`. In the same manner, when the driver inserts a resource `(tagId, "out")` in the bag `Status` it removes all the resources corresponding to the tag in the bag `Area`. This simplifies the application designer's task since she can directly write a rule which starts with a token reading in the bag `Area`.

2) Object `Display`: The second object of the framework manages the displays on the screen. It contains the following bags (non exhaustive list):

- `videoPlayer(playerId, videoname, posX, posY, width, height, orientation, soundTrack)`: this bag accepts only the `put()` operations and launches a video player displaying the video corresponding at filename with the given geometry, with or without soundTrack;
- `video(videoname, status)`: maintains the video status: `started, finished, paused`;
- `videoPlayerCmd(videoname, command)`: accepts only the `put()` operations and the following commands: `"stop"`, `"pause"`, `"resume"`, `"fs_on"`, `"fs_off"` (fs is full screen).

A simple usage of this object is described in Fig. 4. This initialisation rule starts the init video presenting the table on the top left corner of the screen. When the performance is executing, a video player is started and configured to display the video with the resolution (640x480) at position (0,0). The status of the video is set to `"started"`.

```
::
{
 ["Display","video"].put("video_table","started");
 ["Display","videoPlayer"].put("vlc","video_table",
      "0","0","640","480", "True");
}.
```

Fig. 4. Start presentation video of the table

To easily support any kind of video player, the framework uses an external Linux process. The role of this process is to display a video according to a media definition file containing the basic information needed to define the layout, the position, the fact that the sound track is on or off. The display driver saves the PID of the process started in order to interact with it independently of the video player used.

Fig. 5 shows how to stop a video. The precondition waits that the stop card is placed on the table. It then reads the `videoId` of the started video. The performance actually stops the video just by sending the signal `SIGKILL` to the PID playing the video.

```
["Rfid","TagStatus"].rd("tag_stop_video", "in")&
["Display","video"].rd(videoId,"started")
::
{
 ["Display","videoPlayerCmd"].put(videoId, "stop");
}.
```

Fig. 5. Stopping a video with a control card

3) Object `2D engine`: The third and last object of the framework is a 2D engine. It is in charge of displaying the background of the table. It is also in charge of the displayed animations. The current version relies on a Scalable Vector Graphics (SVG) engine to define 2D animations that will be displayed in a simple web browser that is opened in full screen on the table display.

The non exhaustive list of bags is:

- `Background(imagefile)`: When a resource (i.e., an image file) is inserted it replaces the current background of the table;
- `Media(tagId, filename)`: associates a `tagId` to a `filename`;
- `Sprites(spriteId,x,y, svgfile)`: Allows to display a sprite (svg image) at the position `x,y` on the table screen;
- `MoveSpriteGrid(spriteid,x,y,duration, nbsteps,renderlist)`: Allows to define an animation for the sprite defined by `spriteid`. The `duration` of the animation using; `nbsteps` steps and using successively the svg patterns defined in `renderlist`;
- `Visibility(spriteId,percent)`: defines the opacity and the visibility of the sprite.

This object contains in real more than 20 bags that allow not only to define a background but also sprites that can be animated on top of this background. All the svg attributes may be dynamically modified.

The animations are done at the level of the object which returns an html file when invoked through url. This html file is built from static information (templates) present in the file system and contextual information present in the bags. SVG and embedded javascripts take care of the dynamic aspects.

IV. Framework in context

After presenting the framework architecture, this section shows how interactions can be easily encoded with rules, through several examples. It then discusses the interest of the framework compare to other solutions aggregating sensors information. Finally it presents a more complex scenario implemented with the framework.

A. Architecture

Figs 6 presents the current hardware and software setting. It contains the table described in Section II. In addition there are two computing resources: a laptop and a raspberry pi; both embedded inside the table, hidden from the users. The table's screen is connected to the raspberry pi with a HDMI cord,

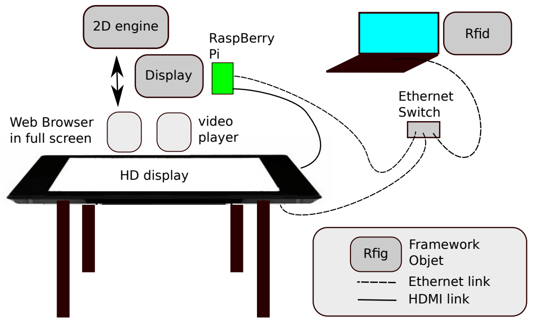

Fig. 6. Global Picture

offering a 1080p HD resolution. The Ethernet switch defines a local area network connecting the matrix of Rfid readers, the raspberry pi and the laptop. From the software point of view, the described objects of the framework are distributed among the two computing resources. The `Rfid` object runs on the laptop, the `2D engine` and the `Display` objects run on the Raspberry pi. In addition a web browser runs on the raspberry pi. It is connected to the `2D engine` object that returns a SVG file according to the current context. The web browser is displayed in full screen on the table display. Video players may be launched on demand on the raspberry pi and displayed on the table screen on top of the web browser.

B. Simple interaction through rules

1) Change the background with a card: In this example (Fig. 7), the background displayed on the screen is changed when a card of type background is put anywhere on the table. The first line of the precondition makes the rule fire only for tags that are known to define a background. Then whenever a card of type background is put on the table (line 2), line 3 finds the filename of the background to render corresponding to the tag id. Then in the the performance phase the action (line 6) changes the background of the table with the content of filename. After the change, it is not necessary to let the card on the table.

```
["Rfid","Type"].rd(tagId, "background")&                      1
["Rfid","TagStatus"].rd(tagId, "in")&                         2
["2D_Engine","Media"].rd(tagId, filename)                     3
::                                                            4
{                                                             5
 ["2D_Engine","background"].put(filename);                    6
}.                                                            7
```

Fig. 7. Rule to change the background when a card is put on the table

You can define as many background as you want, you just need to insert a resource defining the type of your rfid tag as a "`background`" in the bag "`Type`" and a resource to associate the tag id to an image filename in the bag "`Media`".

2) Display a video at the location of the card: This example aims at starting a video when a card of type video is put. The card's position defines the top-left corner of the video. Fig. 8 gives the rule implementing this scenario. As previously, line 1 and 2 make the rules fires when a video card is put on the table. Line 3 gives the card's position. Finally, line 4 finds the video to be displayed from the tag id. The performance then embeds in one transaction:

- ensuring that the card is still there (line 7);

- starting of the video player with (posX,posY) (line 8);
- saving state "started" for the video (line 9).

```
1  ["Rfid","Type"].rd(tagId, "video")&
2  ["Rfid","TagStatus"].rd(tagId,"in")&
3  ["Rfid","Position"].rd(tagId, posX, posY) &
4  ["Rfid","Mapping"].rd(tagId, videoId) &
5  ::
6  {
7   ["Rfid","TagStatus"].rd(tagId, "in");
8   ["Display","videoPlayer"].put("vlc", videoId, posX,
        posY, "640", "480", "True");
9   ["Display","videoPlayer"].put(videoId, "started");
10 }.
```

Fig. 8. Display video at card's position

3) Play a video at the location of the card or in full screen: This scenario uses two rfid cards, one card to start a video where the card is put and one to display the video in full screen. There are three possible conditions:

1) video card only: display the video where the card is put;
2) both cards: display the video in full screen;
3) full screen card only: do not display anything.

Condition 1 has been detailed in Fig. 8.

Fig. 9 implements condition 2. Lines 1-3 check that both cards are on the table. Line 4 fires when the video has been started (by rule in Fig. 8). The performance phase checks that both cards are still on the table, the video is started an the video player is switched to full screen by adding a resource in the bag `videoPlayerCmd` (line 10).

```
1  ["Rfid","TagStatus"].rd("tag_fullScreen", "in")&
2  ["Rfid","Type"].rd(videoId, "video") &
3  ["Rfid","TagStatus"].rd(videoId, "in")&
4  ["Display","videoPlayer"].rd(videoId, "started")&
5  ::
6  {
7   ["Rfid", "TagStatus".rd(videoId, "in") ;
8   ["Rfid","TagStatus"].rd("tag_fullScreen", "in");
9   ["Display","videoPlayer"].rd(videoId, "started");
10  ["Display","videoPlayerCmd"].put(player, "fs_on");
11 }.
```

Fig. 9. Put video in full screen

It is then necessary to write a rule (Fig. 10) to leave the full screen mode if the full screen card is removed from the table. This rule is triggered when the full screen control card is `"out"` (line 1). As previously, the performance checks the player and the cards' status and adds a resource in the bag `videoPlayerCmd` (line 10).

```
1  ["Rfid","TagStatus"].rd("tag_fullScreen", "out")&
2  ["Rfid","Type"].rd(videoId, "video") &
3  ["Rfid","TagStatus"].rd(videoId, "in")&
4  ["Display","VideoPlayer"].rd(videoId, "started")&
5  ::
6  {
7   ["Rfid", "TagStatus".rd(videoId, "in") ;
8   ["Rfid","TagStatus"].rd("tag_fullScreen", "out");
9   ["Display","VideoPlayer"].rd(videoId, "started");
10  ["Display","VideoPlayerCmd"].put(player,"fs_off");
11 }.
```

Fig. 10. Leave full screen

C. Discussion

This section has illustrated the simplicity of writing interactions with the proposed framework. The detailed examples show how information coming from a set of sensors may be aggregated as a complex distributed event.

This is usually implemented with a publish-subscribe approach[7], where subscribers register to specific events generated by publishers (rfid readers in this paper). This has been applied in the context of sensor networks [8]. With such a system it is possible to write code that would be similar to the precondition part of the rules presented in this section. For instance to react to a tag detected in a specific area or to an external event. However, in a publish-subscribe approach when the system has to react upon a set of events or to be sure that the events are still valid when the actions have to be executed, the amount of additional code is not negligible.

With the framework developed on top of our middleware expressing an event as "one card is put in a specific area" and "another card of a specific type is put at the same time anywhere else" is simply a sequence of rd() tokens.

In addition, defining what to do if a card is put on the table an immediately removed is possible thanks to the distributed transaction offered in the performance phase.

D. Scenario

We described here a simple application based on this framework which uses quite complex interactions between several users around this table. The application allows to collect the opinion of a panel of people to elect the better equipment, concept or decision according to a set of criteria. In the present example the panellists are asked to give their opinions about a set of smartphones according to the following criteria: aesthetic, user interface, size and autonomy. These criteria are noted respectively A, B, C and D and can take the value positive or negative depending on the majority of vote from the panellists. The resulting information is displayed as a Veitch diagram as shown in Fig. 11.

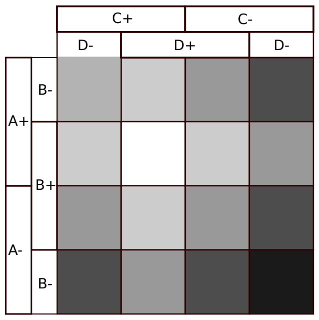

Fig. 11. Veitch Diagram

The white cell contains the best choices that receive 4 positive opinions. The adjacent cell contain choices that receive 3 positive opinions. The darker a cell is, the more negative it is. The black bottom right cell contains the worst choice with 4 negative opinions. This section now details what is an interaction session and gives a few hints on the implementation.

1) Interaction session: At the beginning, each panellist has a badge representing her identity. The master of session presents a smartphone and may display a video on the table by putting the corresponding card on it. Some modifier cards added to the table may modify the display either by switching to fullscreen or by launching a second video player with a 180 rotation to adapt to situation where people are all around the table. In this case, the second video uses the same video flow (without sound track) and is synchronised with the first one. Once the presentation is done, the vote may start. The master of session places the card corresponding to the criterion (e.g., aesthetic) and then the table display shows two areas, one green to collect the badges of panellists liking the smartphone design and one red for those that are not enthusiastic. An additional video or photo specific to this criterion may also be displayed. Then, the master of session triggers the vote by placing an hourglass (tagged with an rfid) on the table. A timer indicates at each corner of the table the remaining time for the vote. Each panellist put her badge on the table according to her opinion. A circle is displayed around the badge to return a feedback to the user. Different modalities may be configured at the beginning of the session to control the vote:

- the duration of a vote phase;
- missing vote is considered as negative or positive;
- a vote is definitive or not.

Once the timer reaches zero the votes are stored for further processing and the master of session can go to the next criterion. When all the criteria have been considered, the master of session can go to the next smartphone. At any moment, the master of session may place a card on the table to display or print current status of the Veitch diagram.

2) Implementation: The full application described here may be implemented by using small variation of the basic interactions involving `Display`, `2D_Engine` and `Rfid` objects presented in the framework section. plus a specific `Veitch` object that contains bag used to store panellists identities, votes and current step in the session (smartphone number and criteria number). The basic settings, configuring a working session, are done through initialisation rules that define modalities such as default value of missing vote or the identity of the panellists. Note that using the proposed framework is very appropriate since adding new features simply requires to add new rules. Existing rules can continue to work without concern. We can for instance, use an initial round getting the identities of the panellists rather than using a configuration rule. This can be done without any other impact that replacing the initialisation rule with the three rules required to obtain the identity of the panellists: i) open identity round with detection of the specific control card, ii) copy each badge id detected to the bag panellist and iii) close identity round by detecting that the specific control card is no longer on the table. This obviously works, with any number of panellists.

V. CONCLUSION

This paper has presented an innovative hardware and a framework easing the development of applications on top of it. The hardware combines a full HD display and a set of 384 rfid readers allowing to return the location of several tens of object tagged with Rfids.

The framework is built on top of a rule-based middleware relying on production rules and distributed transactions to ease events' combination handling. It uses a driver which maps rfid events into resources stored in bags allowing the resources to be accessible with simple rules, thanks to the middleware. This allows to react to event composing several rfid tags and to embedded event verification in distributed transactions.

To ease the developer task, bags are grouped into three objects to manage the Rfid reader matrix, to start and stop videos and to interact with the 2D engine offering the dynamic background of the table. This paper has shown through simple examples and a more complex scenario how the framework helps designing applications for the proposed hardware.

Future works will take two directions. One is to develop other applications around the decision making field to show that the table plus the framework is a full kit to quickly develop and customize such type of applications. The second is to integrate more external sensors and actuators to the framework. Cameras which can deduce the number of people around the table (e.g., counting the detected faces), sensors computing the distance of the users from the table, voice interface, etc. are informations that once combined with information returned by the table may offer a richer user experience.

ACKNOWLEDGEMENT

The authors would like to thank Nicolas Géraud from Dasein Interactions for the numerous advices about the user's interactions of the Veitch application. This work has been partially funded by the FP7 SCUBA project under grant nb 288079 and FUI Rapsodie project under grant nb F1209039V.

REFERENCES

[1] L.-F. Ducreux, C. Guyon-Gardeux, S. Lesecq, F. Pacull, and S. R. Thior, "Resource-based middleware in the context of heterogeneous building automation systems," in *IECON 2012-38th Annual Conference on IEEE Industrial Electronics Society*. IEEE, 2012, pp. 4847–4852.

[2] J.-M. Andreoli, F. Pacull, D. Pagani, and R. Pareschi, "Multiparty negotiation of dynamic distributed object services," *Journal of Science of Computer Programming*, vol. 31, pp. 179–203, 1998.

[3] D. Arregui, C. Fernström, F. Pacull, G. Rondeau, and J. Willamowski, "Stitch: Middleware for ubiquitous applications," in *In Proc of the Smart Object Conf*, 2003.

[4] N. Carriero and D. Gelernter, "Linda in context," *Commun. ACM*, vol. 32, pp. 444–458, April 1989.

[5] T. A. Cooper and N. Wogrin, *Rule Based Programming with OPS5*. Morgan Kaufmann, July 1988.

[6] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency control and recovery in database systems*. Boston, MA, USA: Addison-Wesley Longman Publishing, 1987.

[7] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM Computing Surveys (CSUR)*, vol. 35, no. 2, pp. 114–131, 2003.

[8] E. Souto, G. Guimarães, G. Vasconcelos, M. Vieira, N. Rosa, C. Ferraz, and J. Kelner, "Mires: a publish/subscribe middleware for sensor networks," *Personal and Ubiquitous Computing*, vol. 10, no. 1, pp. 37–44, 2006.