User Interface Distribution in Multi-Device and Multi-User Environments with Dynamically Migrating Engines

Luca Frosini
HIIS Laboratory – ISTI-CNR
Via G. Moruzzi, 1
56124 Pisa (Italy)
luca.frosini@isti.cnr.it
+39 050 621 2602

Fabio Paternò
HIIS Laboratory – ISTI-CNR
Via G. Moruzzi, 1
56124 Pisa (Italy)
fabio.paterno@isti.cnr.it
+39 050 621 3066

ABSTRACT
In this paper we present a framework and associated run-time support for flexible user interface distribution in multi-device and multi-user environments. It supports distribution across dynamic sets of devices, and does not require the use of a fixed server. The distribution updates are processed taking in account device types and user roles. We also report on three example applications and a validation of the presented framework.

Author Keywords
Multi-device User Interfaces, Development Tools, Distributed and Migratory User Interfaces.

ACM Classification Keywords
H.5 Information Interfaces and Presentation; H.5.2 User Interfaces, H.5.3 Group and Organization Interfaces.

INTRODUCTION
In the last decade a wide variety of interactive devices have penetrated the mass market, and people spend more and more time using them. This has made it possible to create many environments where people spend a long time interacting with various devices sequentially or in parallel [4].

In order to better exploit such technological offer often people would like to better use multiple devices while interacting with their applications, for example to dynamically move components of their interactive applications across different devices with various interaction resources.

Unfortunately, the development of multi-device user interfaces is limited by current interaction development toolkits, which are still designed assuming to support the development of user interfaces for single devices without providing support for multi-device access. At the research level some frameworks for multi-device user interfaces have been proposed but usually their support has been limited to specific contexts and applications, and thus their adoption has been rather limited.

The framework we present provides developers with an API that can be exploited both in Web and Java applications in order to obtain more easily application user interfaces (UIs) that can be dynamically distributed and/or migrated in multi-device and multi-user environments. The framework also allows dynamically creating multiple simultaneous sessions for applications used by groups of devices where the UI is distributed. Furthermore, it does not require a fixed server to manage the distribution. The elements of the UI can be distributed by specifying specific device(s), group(s) of devices, specific user(s), and groups of users according to roles.

In the paper, after discussion of related work we provide a description of the architecture exploited by our framework, the main concepts that characterize it, and the associated possible commands. We then describe three multi-device applications developed with it that have different requirements and discuss its generality and performance. Lastly, we draw some conclusions and provide indications for future work.

RELATED WORK
Distributed User Interface (DUI) is a topic that has been addressed from various viewpoints. The main aspects of user interface distribution are indicated in [1]: *What* can be distributed, *When* the elements can be distributed, *Who* can distribute and *Where* they can be distributed. Other important aspects of distributed user interface are *Portability*, *Decomposability*, *Simultaneity* and *Continuity*. These, as argued by Peñalver et al. [9], are needed properties to be guaranteed in a DUI and our framework aims to address them.

Furthermore, distributing user interfaces without constraints can result in an unusable user interface as argued by Luyten et al. [2]. A label distributed in a device and the correspondent input field in a different one is a simple example of unusable user interface. In order to address such issue we provide the possibility of specifying what elements can be distributed.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
EICS 2014, June 17–20, 2014, Rome, Italy.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2725-1/14/06..$15.00

One contribution in this area provides a proposal for a peer-to-peer solution [3]. One issue in this regard is the lack of an explicit component able to maintain the state of the distribution at any time. In our solution we provide such component, which can be located and moved to any device involved in the distribution.

Another contribution [6]proposes a collaborative environment for the distribution of applications useful to support some tasks. A framework to orchestrate the spanning of a web-based UI over many different screens has been proposed by Hartmann et al.[7]. In contrast, our framework focuses on the distribution of user interfaces in multi-user and multi-device contexts in such a way as to limit the impact in the application code in order to also easily obtain distributed user interfaces in existing single device applications. A platform supporting distributed application user interfaces on interactive large public and personal mobile device screens has been proposed [8]. Our proposal is based on a similar session concept, though it also enables creating shared sessions amongst mobile and fixed devices.

A catalogue of distribution primitives to orchestrate DUI screens has been proposed [5]. In our proposal we use two simple commands maintaining the same expressivity. This decreases the time for developers to learn the framework and facilitates code reusability.

Fisher et all [10] describe general challenges for P2P DUI development in terms of design, architecture, and implementation but they do not provide a framework for the development of distributed user interfaces.

While previous work [11] has considered the use of model-based languages for supporting distributed user interfaces, in this work we have aimed to identify a solution with good performance that can be exploited in various applications domains. Thus, we have considered previous research [12] and report on a novel solution that is more flexible in terms of management of the distribution state, with the ability to exploit dynamic sets of interactive devices, and report on a validation in terms of applications developed, performance, and analysis of the impact on the code.

THE ARCHITECTURE OF THE FRAMEWORK

Our framework is logically composed of a library and runtime support. The library is used by the developers to introduce UI distribution in their applications. The runtime support can run on a dedicated server or in one of the devices participating in the distribution.

Figure 1 shows the logical components of the framework and its run-time support. There are two main blocks: Engine side and Client side.

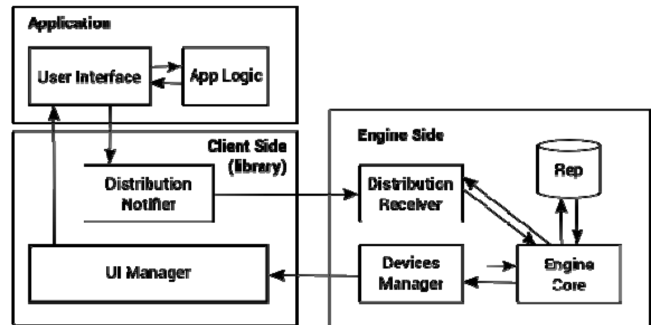

Figure 1. Overview of the framework architecture.

The Engine side is the runtime support and is responsible for managing the requests of distribution changes, processing them, and calculating the new distribution state.

The *Engine* is able to maintain the *Current State* of Distribution, which allows devices to join a distribution session at any time and sets their UIs in the proper state. The distribution state is mainly based on the concept of UI elements State. We have identified three states: *Invisible* (element is not visible at all), *Disabled* (visible but not reactive to user actions, e.g. a button that does not react to the users' clicks) and *Enabled* (visible and reactive to the associated user-generated events). Thus, it is possible to define a simple relation across the states, through which each state adds some aspect to the previous one (disabled adds visibility to the invisible elements; enabled adds reactivity to the disabled elements).

The *Client Side* library represents the component responsible of sending distribution change requests to the engine, and receiving updates of changes to apply on the managed UI. On the client side there is a *UI Manager* that receives and processes the notifications of the distribution updates arriving from the engine. The *UI Manager* invokes a callback function in order to apply the received update..

The callback is triggered in response to a distribution update received from the engine (because of a state change or a value change of a UI element). The framework provides some standard behavior for each change. Moreover, the framework provides the possibility of specifying personalized behavior to apply when a distribution change occurs. For example, the default callback for an *ASSIGN Notification* to set an element to invisible simply makes the element no longer visible without any additional effect. If the developer wants to use a fading effect when this occurs, he can register a specific callback for this purpose.

Figure 1 also shows the application part which uses the framework client side.

Please note that the *Client Side* library does not make any distribution choice, instead such changes depend on the application. In the application any policy can be implemented.

Authentication Process

When a device wants to take part in the distribution it has to subscribe to the engine. The device sends its own capabilities and credentials to the engine. Using the supplied credentials the engine decides whether to allow the device to take part in a session. Furthermore, according to the supplied capabilities the engine inserts the device in one or more groups. The groups are used in the distribution to target devices with similar capabilities.

Once the device is allowed to take part in a session, the engine sends information regarding what UI elements should be shown to the user. We refer to this information as the *Distribution State* (as explained in the next section).

A device (if it has proper credentials) can subscribe to a distribution session at any time. Each device receives from the engine the distributed UI consistent with the other devices, and in the same situation as if it had subscribed at the beginning of the session.

During the session life the device can send updates of the UI distribution to the engine or receive notifications by the engine of updates made by other devices.

Any communication made by the device to the engine is accepted by the latter only if the device has enough rights.

Distribution State

The framework is based on the concept of distribution state, which indicates what UI elements are associated with each device and their state. Indeed, the framework allows developers to assign three basic states to the user interface elements: enabled, disabled (which means visible but not reactive to events), and invisible. We can express the relations: *Enabled* > *Disabled* > *Invisible*

At any time the distribution state of a UI is known by the engine.

The state changes dynamically under the effect of updates generated by clients and elaborated by the engine.

The component that knows the distribution state is not in a fixed server but can be in any device involved in the distribution, even in a mobile device.

When the engine receives an update it performs these main operations:

- Validate the request;
- Calculate the new *Distribution State*;
- Calculate which devices have to be updated;
- Inform involved devices of changes in their UIs.

Devices categorization (Target)

The framework exploits two concepts (*Type* and *Role*) to address the devices involved in the distribution.

Type

The type concept is associated with a set of device capabilities. The possible types are not static but they can vary depending on the application.

When a device subscribes to the distribution it provides its capabilities and the engine assigns one or more types to the devices. The *Type* is used by the engine and by other devices to address a group of devices without the need to know devices ID.

In many cases we can find that some types are a subset of another. For example a Laptop is a sub-type of PC because requires all the capabilities of PC plus a Battery. We can argue that the PC type is broader (or at least equal) to the Laptop type. All Laptop devices will also be considered PC devices but not vice versa. In this case we can say that PC > Laptop. Two sets are not comparable if one is not a superset of the other.

The possibility of comparing different types is important when the engine receives a distribution update: first of all to calculate the new *Distribution State* (reducing it); secondly to avoid sending unneeded updates to devices.

The reduction of the types enumerated in the state (when possible) implies a more efficient calculation for managing states.

Role

The role concept is related to the type of tasks carried out by the user using a device in the distributed application and is independent of the device type.

In order to exploit the distribution support the user authenticates through an identity certificate, which contains information regarding the user role. The details regarding how certificates are issued, verified and implemented are beyond the scope of this paper.

The Distribution API

The framework supports an API with two commands (from clients to engine) to perform distribution changes. The first (*ASSIGN Command*) is used to change the devices that can display or allow manipulation by the user of a certain element in the UI. The second (*Feedback Command*) informs devices of a change in the value of an element. Examples of the feedback that can be provided is that of the values of an input field, the selected tabs in a tab container, the center and the zoom values of an image which can be panned or zoomed.

ASSIGN Command

The parameters of the ASSIGN command are:

- *What*: identifies an interface part, typically the ID of the element or the container of elements that has to be distributed.

- *Target*: specifies the devices that should receive it, they are indicated by type(s), role(s) or identifier(s).
- *Basic State Level*: identifies the new state levels for the elements indicated in What for the devices specified in Target.

Feedback Command

The parameters of the Feedback command are:

- What: identifies an interface part, typically the ID of the element or the container of elements to be considered.
- Data: indicates the new value for the element identified by What that will be sent to all the devices with a State Level > Invisible

These commands flow from client to engine. Once the engine has elaborated them it will generate a corresponding command flowing from the engine to the involved clients. We will refer to them as Notification commands.

ASSIGN Notification

This is the engine's *ASSIGN Command* response. Because the notification is sent directly to the device, the target is implicit and for this reason it is omitted.

- *What*: identifies an interface part, typically the ID of the element or the container of elements that has to be distributed.
- *Basic State Level*: identifies the new state levels for the elements indicated in What for the devices considered.

Feedback Notification

This is the engine's Feedback command response and has the same content and signature:

- What: identifies an interface part, typically the ID of the element or the container of elements to be considered.
- Data: indicates the new value for the element identified by What that will be sent to all the devices with a *Basic State Level > Invisible.*

Distribution Orchestration

The framework has been designed in order to support different sessions for the same application. Depending on their role users can:

- Create new sessions;
- Subscribe to an existing session;
- Leave a session;
- Subscribe to all sessions;
- Unsubscribe from all sessions;
- Send ASSIGN command to distribute an element;
- Send Feedback command to change the data value of an element.
- Manage the devices subscribed to a session.

When a device subscribes successfully to the distribution environment it receives the *Current State* of the UI. The *Current State* is communicated through an array of *ASSIGN Notification* and *Feedback Notification* containing all the information needed by the client to update its own UI.

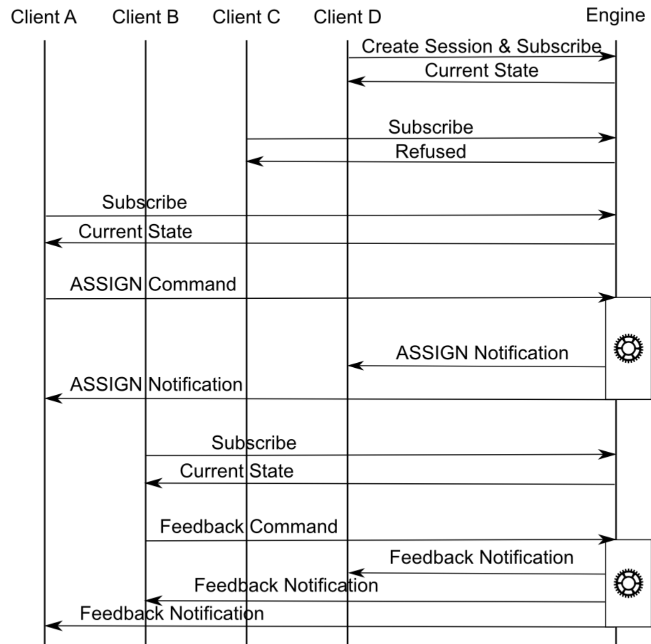

Figure 2. Sequence Diagram describing examples of subscription requests (accepted and refused) and commands sent by multiple devices.

Figure 2 shows a sequence diagram describing an example of interactions between the engine and four different clients that subscribe and then send distribution updates. Client D creates a session and subscribes itself. After that client C tries to subscribe but the request is refused by the engine because it does not provide the necessary credentials. Client A instead subscribes successfully and then sends an *ASSIGN command*. The engine calculates the new state and sends the corresponding notification to the involved devices. This is repeated by client B, which after a successful subscription sends a *Feedback Command*.

A device can request to be part of all the possible sessions (if this is allowed by the configuration parameters of the application). Only when the device with the relevant rights activates it then it can be actually part of a specific session. When a device is removed from a session then it gets back in a waiting list for participating in another session.

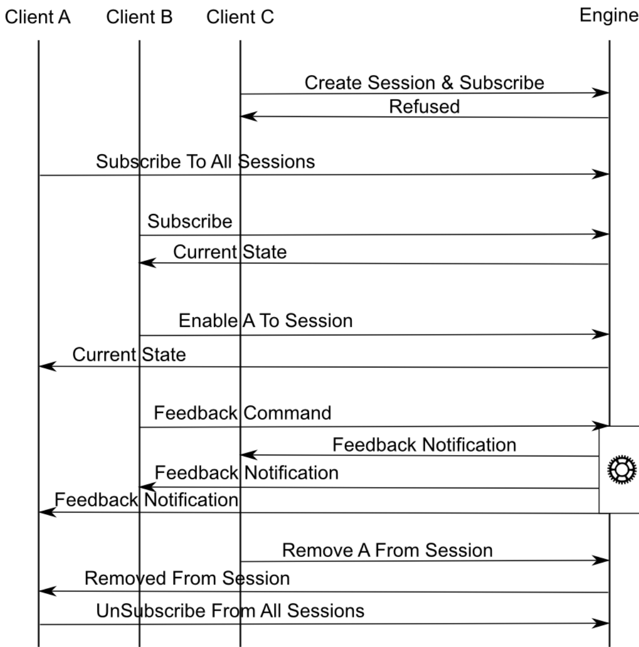

Figure 3. Sequence Diagram describing an example subscription and all related requests.

Figure 3 shows a sequence diagram describing an example of subscription to all session from a client (A) and then client (B) enables it a specific session.

A device that has subscribed to a session then receives all the relevant UI updates. When the clients receive the updates they apply them according to their UI element types.

The distribution of all the standard UI elements supported by Web and Android user interfaces is provided for through an extension mechanism using callbacks. This is useful in case the developers want to customize how the corresponding state is updated, or to achieve specific effects in updating the state (e.g. introducing a fading effect when an image disappears).

Distribution State Representation in the Engine

For each distributable element of the UI a JSON document is created and memorized in a document-oriented database (CouchDB). When the *Engine* receives an *ASSIGN Command* for a UI element, the groups indicated in the *Target* parameter are compared with groups associated to each *Basic State* of the element. The element states are updated taking in account any relations existing among groups. Let us suppose that a button is *Invisible* for the *Smartphone* devices and an *ASSIGN Command* to set the button to *Disabled* for *Mobile* devices arrives at the *Engine*. If we suppose that the *Mobile* type includes the *Smartphone* type then the *Smartphone* type is no longer associated with the *Invisible* state, and the *Mobile* type is associated with the *Disabled* state for the button.

Analyzing more in detail how the state is maintained, we can see that for each user interface element there is an indication of the corresponding targets for each of the three possible basic states (Figure 4 shows an example). The table contains an assigned sequential number for each possible target. This sequential number is incremented by the engine each time it performs a state update. Thus, these numbers define a temporal order between the elements of the distribution state table.

For example, by analysing the state presented in Figure 4 it is possible to understand that for the element with *TabHost* id, the *Mobile* devices visualize it but they cannot interact with it since they are associated with the *Disabled* state (and are associated with number 2), while all the other devices visualize the element and can interact with it.

Furthermore, all the devices associated with users with the *Admin* role visualize and can interact with *TabHost* even if the device is *Mobile*. In fact, the number for the *Admin* role (5) is higher than the Mobile one (2) and this means that the corresponding state change occurred afterwards, and is thus the currently dominant one.


```
{
    "_id": "TabHost",
    "type": "element",
    "Enabled": {
        "types": {"ALL":0},
        "roles": {"Admin": 5},
        "devices": {}
    },
    "Disabled": {
        "types": {"Mobile":2},
        "roles": {},
        "devices": {}
    },
    "Invisible": {
        "types": {},
        "roles": {},
        "devices": {}
    },
    "data": {
     "position": 0
    },
    "count": 6
}
```


Figure 4. Distribution State representation of a UI element.

Moving the Engine

As we mentioned the Engine can be moved dynamically. For this reason the framework provides an API to move the *Distribution State*.

If the engine has to be moved from one device to another, the receiver device invokes the *MoveEngine Command* in the current *Engine*. This command contains as parameter the URL where the new *Engine* will be available. If the requester has sufficient rights to become the new *Engine*, the current one serializes the *Distribution State* and sends it to the requester device. The *Distribution State* is sent to the new *Engine* in a notification called *Distribution State*

Notification. Please note that this is not the *Current State* (which contains only the State for a Device) but the full *Distribution State*.

Then, the old *Engine* stops processing other requests from devices and sends an *EngineMoved Notification* (with the URL of the new *Engine*) to the devices already subscribed to the session. If a device wants to continue to participate in the distribution, it must subscribe to the new *Engine*. The callback associated with the client for the *EngineMoved Notification* is already implemented for subscription to the new *Engine* and developers need do nothing unless they want to personalize the behavior.

APPLICATIONS

In order to verify the suitability of the framework we have used it in the design and development of three applications that needed distributed user interfaces in three different contexts of use.

Museum Guide

One application aimed to improve the user experience during a museum visit. So, we considered a single user application able to exploit mobile devices in conjunction with a public display in an indoor environment. The museum has some large public displays, which allow visitors to access relevant multimedia content. When users are near the large screen they can use the smartphone to select the content shown in it. For example, the visitors can select and display some high-resolution images of artworks that for some reasons (e.g. security, or art preservation issues) cannot be viewed from a short distance.

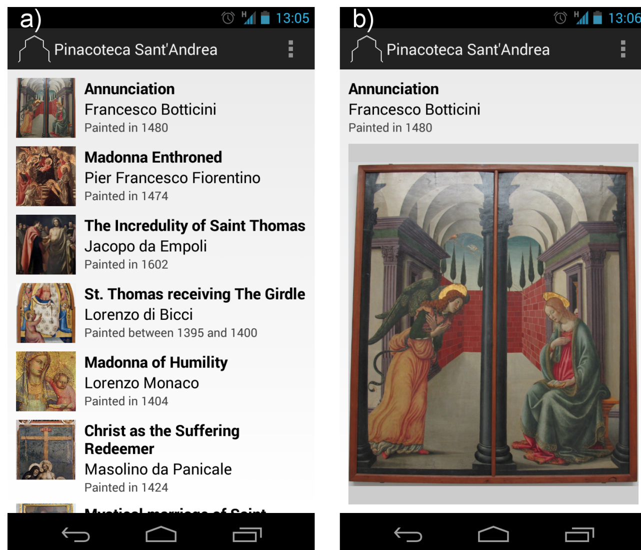

Figure 5. User interface of museum Android application. The one on the left (a) shows the artwork list and the one on the right (b) shows the artwork picture draggable and zoomable.

In addition, it is possible to access graphical information regarding the available artworks, such as maps indicating where the artwork was located before arriving at the museum, sketches of the artwork made by the artists, when available, and video guide introducing the opera.

When started the application on the large display side (implemented using the web version of the framework) immediately subscribes to all the existing sessions through the framework.

Users can launch the application through Android devices, which immediately create and subscribe to one session. When they want to access the large screen they use the smartphone's camera to scan the QR code with the ID of the large screen, which is used to add it to the session and then allow interaction.

On the smartphone it is possible to access the artworks list and select one artwork of interest through touch (Figure 5a). The device will generate a *Feedback Command* containing the URI of the image and default zoom and center for the image.

Then the users can perform pan and zoom through their smartphone in order to control what is visualized in the large screen (Figure 5b).

Each time a user zooms or pans the image on the mobile a *Feedback Command* with the new center and zoom level is sent to the *Engine*. When the large screen receives the *Feedback Notification* it will apply it. On the large screen a personalized callback is used to correctly apply the new zoom and center.

The data value of the *Feedback Command* and *Notification* have the following structure:

```
data : {
        URI: “VALUE”,
        zoom: 0,
        center : [960.0,540.0]
}
```

Using the menu (Figure 5b top-right corner) is possible to show in the large screen one of the tabs (white background bottom center in 6a and 6b) which contains the extra content of the artwork.

Selecting one of the menu options, an ASSIGN Command with the id of the selected element (tab on large screen) is generated. The Basic State Level will be Disabled and the target will be the large screen ID. When it receives the Feedback Notification, the large screen will show the tab. When the user returns to the image, an ASSIGN Command to set the tab to Invisible will be generated. The large screen implements its own callback to perform the desired behavior.

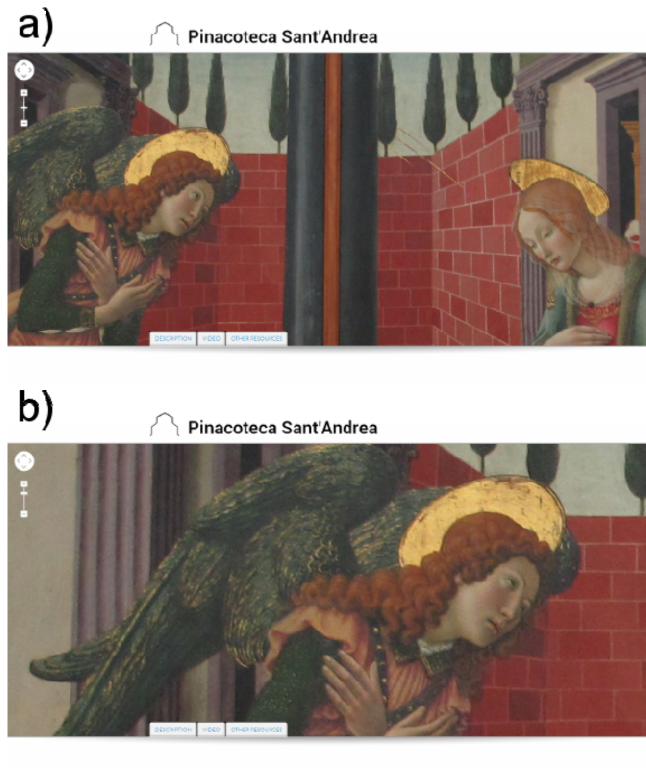

Figure 6. User interface of large display application. 7a (above) shows a lower level of zoom than the 7b (below) and a different center position.

Once the users have accessed an artwork, they can continue the navigation to other items or release the large screen. In any case, after some time without user interaction there is a timeout event that allows the release of the large screen from that session, which occurs also if the user starts to interact with another large screen.

City Guide

The second application is a city mobile guide support for groups of visitors. So, it is an example of a multi-user, multi-device application for outdoor environments.

The application supports guides accompanying groups of visitors who can have either tablets or smartphones. The application shows information supporting the mobile visit. The application version associated with the guide role allows them to select the content to show in the version of the tourist role.

Thus, the user interface elements are in the *Enabled* state for the guide while they can be *Disabled* or even *Invisible* for the tourists. The guide can interactively change the states for the elements shown in the tourist's user interfaces.

The guide has the rights to create and subscribe to a session while the tourists can only subscribe to the session created by the guide. This is achieved through the certificate mechanism previously introduced, which allows the environment to provide different rights depending on the user's role.

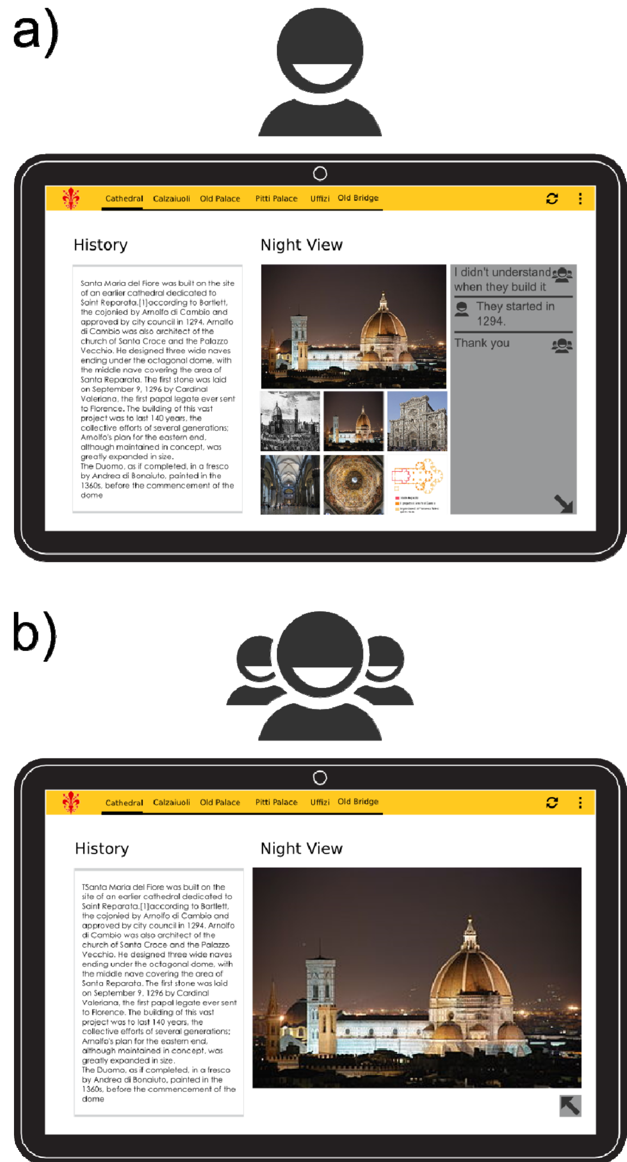

Figure 7. User interface of Android city mobile guide. The image above (a) is the guide version and the one below (b) is the tourist version.

Figure 8a shows the city guide application. The guide can select the different images to be shown to tourists. The big image (element with id *FeaturedImage*) is visible to tourists and enabled to guide. The thumbnails block is enabled to guide and invisible to tourist. Tapping the thumbnail the guide change the *FeaturedImage*. Thus, a *Feedback Command* is sent to the *Engine* containing the URI of the image to be shown.

Crossword Game

The last application is a crossword game that allows multiple users to participate in solving a puzzle with the support of a PC or a WebTV.

Users subscribe to the session through their mobile devices and are thereby able to enter words in the crosswords. Each time that a user enters a word then it is shown on all the other users' devices.

When a user insert a word a *FeedbackCommand* is generated so all the other involved devices will be updated about the solved word.

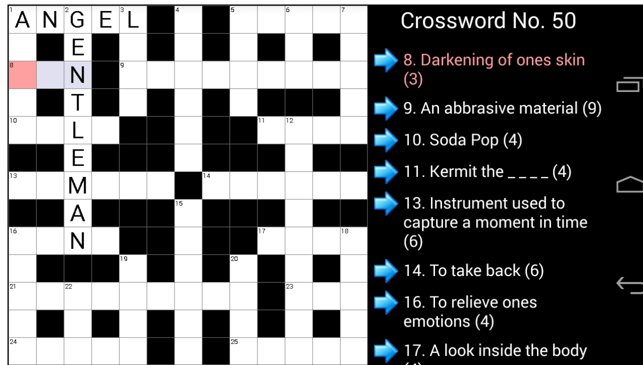

Figure 8. User interface of Android crossword application.

Each question is considered an element of the interface and *FeedbackCommand* is used intensively when:

- A user enter a solution;
- The entered solution conflicts with an intersecting word;
- A user suggests a solution to other participant.

The data value has the following structure.

data : {

value: "VALUE",

proposed: [proposed1, proposed2],

discarded : [discarded1, discarded2]

}

Every time a user enters a solution the typed value is inserted in value field. When a user is not sure about a solution but wants to try to suggest it to other users the proposed word is inserted in proposed field and a *FeedbackCommand* is sent.

Furthermore, the user has the possibility to discard a value or a proposed solution. In the default modality all the users have all words elements to be solved in in enable mode.

The device which creates the session is the only one which can distribute the elements using *ASSING Command*. All the others can only send *Feedback Command*.

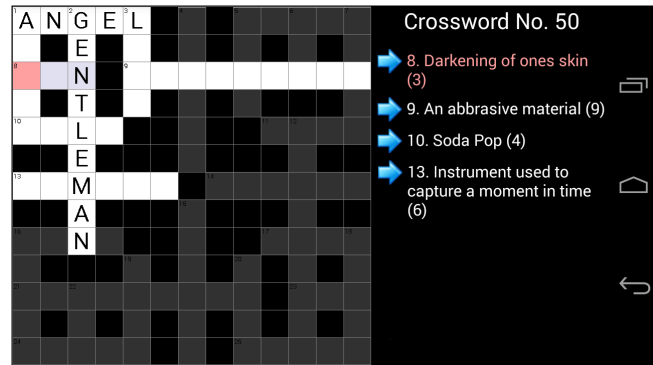

Figure 9. User interface of Android crossword application with hidden questions.

This allows the session creator to switch from default to a version where each user only sees a set of random questions. The creator can redistribute parts to each player with two options:

- with timeout
- manually

In the timeout version the questions are redistributed each time the configurable timeout is triggered. This is performed by the session creator client sending a list of *ASSIGN Command* to the *Engine*. In the second one, it is the user who creates the session that manually triggers the event, the distribution logic of the random questions is completely in the application, no choice is made by the *Engine*.

Figure 8 shows a screenshot of crossword application where all the questions are available to all users. Figure 9 shows instead a screenshot were only 1, 8, 9, 10 and 10 across words and 1, 2, 3 down words are enabled and all the rest are invisible.

To implement this game we used personalized versions of callback both for *ASSIGN Notification* and *Feedback Notification* associated to question elements.

IMPACT FOR DEVELOPERS

In this section we provide a concrete description of the support provided by the framework for the developers, showing how it requires limited number of code instructions.

Indeed, if developers want to start using the framework in order to include a device in the distribution environment using the standard callback functions they just need to write about 40 lines of code. The Android version is shown in Figure 10. With such code the corresponding device will receive all the relevant distribution change notifications.

```java
private void subscribe() throws Exception {
    DistributionClientManager.setApplicationID("APP-ID");
    DistributionClientManager.setSessionID("SESSION-ID");

    distribution = DistributionClientManager.getInstance();
    distribution.setActivityRID(this, R.id.class);

    UUID webSocketKey = UUID.randomUUID();

    List<ClientConnector> clientConnectors = new ArrayList<ClientConnector>();
    clientConnectors.add(new WebSocketClientConnector(webSocketKey));

    List<Feature> features = new ArrayList<Feature>();
    features.add(new Feature("FEATURE 1"));
    features.add(new Feature("FEATURE 2"));
    features.add(new Feature("FEATURE 3"));

    String name = AndroidUtility.getDeviceName();
    String id = AndroidUtility.getDeviceID();

    Device device = new Device(name, id, clientConnectors, features);
    distribution.setRuntimeDevice(device);

    WebSocketEngineConnector engineConnector = new WebSocketEngineConnector(
            webSocketKey, webSocketURL, new SocketOpenCallBack() {
                @Override
                public void call() {
                    subscribeDevice();
                }
            }
    );
    distribution.setEngineConnector(engineConnector);

}

private void subscribeDevice() {
    try {
        distribution.subscribeDevice();
    } catch (Exception e) {
        logger.e(String.format("Error while subscribing device : %s",
                e.getLocalizedMessage()));
    }
}
```

Figure 10. Example of Java code for Android used to subscribe a device to a session.

In Figure 10 the Android code to subscribe to a session is shown. In the first two rows the identified of application and the session is set. A Distribution Client Manager (the Java class wich coordinates all the distribution operations) is instantiated. The client register itself to the istance of DistributionClientManager, with name, id and the list of the capabilities (Feature class) thata the Engine will use to associate one or more *Type* to the Device.

The invocation of suciption is made using the *distribution.subscribe()*; instruction.

The client is using a WebSocket to send command to the Engine and viceversa for the Engine to send Notification, for this reason we can recognized WebSocketEngineConnetor and WebSocketClientConnector. This is an implementation issue out of the scope of this paper

Furthermore, with just 10 more rows of code they can manage the user actions that generate a distribution update to send to the engine. Figure 11 shows an example of code written in JavaScript for the management of Web tabs. The code shows an example of callbacks for *ASSIGN* (*tabHostUIUpdate*) and *Feedback* (*tabHostFeedback*) *Notifications*. These callbacks have to be registered with the *UIManager*.

```javascript
function tabHostFeedback(connector, feedback){
    console.log(feedback);
    var selector = connector.getClientSelector(feedback);
    var position = feedback.data.position;
    console.log("Tab to select : " + position);
    var disabled = jQuery("#TabHost").isDisabled(position);
    var visible = jQuery("#TabHost").isVisible(position);
    if(visible){
        if(disabled){
            jQuery("#TabHost").enableTab(position);
        }
        jQuery(selector).tabs({ active :  feedback.data.position });
        if(disabled){
            jQuery("#TabHost").disableTab(position);
        }
    }
}

function tabHostUIUpdate(connector, uiupdate){
    console.log(uiupdate);
    var position = parseInt(uiupdate.what.id.substring(3)) - 1 ;
    var visible = uiupdate.level > 0;
    var enabled = uiupdate.level > 1;
    if(visible){
        jQuery("#TabHost").enableTab(position);
        if(!enabled){
            jQuery("#TabHost").disableTab(position);
        }
    }else{
        jQuery("#TabHost").disableTab(position, true);
    }
}
```

Figure 11. Example of JavaScript code of callback used to manage a *TabHost*.

VALIDATION

Expressivity

In some interviews developers unfamiliar with distributed UI stated a preference for a limited number of commands for managing distribution. Thus, this was an initial design requirement for our framework. Despite the simplification we aimed to preserve the command expressivity in such a way to be able to provide the same possibilities as presented in previous work [5]. For example, the DISPLAY operation in [5] can be achieved with the ASSIGN command of state level > Invisible; conversely, an UNDISPLAY can be obtained with a state level = Invisible. The MOVE operation can be achieved using an ASSIGN command of state level to Invisible for the elements in the source device and an ASSIGN command of state level to Enabled in the target device.

The REPLACE operation can be achieved by the composition of an *ASSIGN Command* of state level = *Invisible* to the element to be replaced and with an *ASSIGN Command* of state level to *Enabled* to the replacing element.

The MERGE command can be achieved using an *ASSIGN Command* of state level to *Enabled* to the device where we want to merge.

The SEPARATE command can be obtained by using an *ASSIGN Command* of state level to *Invisible* for the elements in the device where the UI is being separated and an *ASSIGN Command* of state level to *Enabled* in the device the element is going to appear on.

Performance

In order to assess the performance of our environment we have focused on the engine performance, and we have measured on average the number of commands it received from the clients during a session, and the minimum, maximum, average, and variance of the time taken to process them. In the processing time we included both that required to calculate the new distribution state but also that required to notify the involved devices of the distribution change.

The results were calculated considering 18 sessions of the city guide application involving three devices gathered during a user test. This application mainly uses the Feedback command because it mainly supports a kind of co-browsing across multiple devices. The average number of commands per session was 287. The minimum amount of time to elaborate a request was 236 msec and the maximum was 661 msec. The average time was 296.46 msec and the standard deviation is 50.11.

With the applications and their informal use we have learnt that the framework processing time is sufficiently short to avoid creating particular usability issues.

CONCLUSIONS AND FUTURE WORK

We have presented a framework and run-time support for enabling cross-device interaction. The client side part is currently available for Android and Web-based applications, and we plan to provide a version for iOS as well.

A number of multi-device user interface applications have been designed and developed through it. We have also shown how the impact of the framework in the application code is limited. In terms of performance we have reported the results of a first study, which are encouraging.

The three applications show that the framework can be used in different domains and in different environments: single or multi-user applications; indoor or outdoor environments; mobile and stationary devices.

Future work will be dedicated to introducing more flexible mechanisms for selecting user interface elements, security, and dynamic device allocation across multiple sessions.

ACKNOWLEDGMENTS

We thank the IUDSM (Distributed User Interfaces and Mobile Security) Project (funded by Regione Toscana, CNR-ISTI and IIT, and Softec)[1] for supporting this work, and Zeno Amerini (Softect) for useful discussions on the topics of the paper.

REFERENCES

1. Demeure, A., Sottet, J.-S., Calvary, G., Coutaz, J., Ganneau, V. and Vanderdonckt, J. The 4C Reference Model for Distributed User Interfaces. ICAS (2008), 1-10.
2. Luyten, K., Van den Bergh, J., Vandervelpen, C. and Coninx, K. Designing distributed user interfaces for ambient intelligent environments using models and simulations. Computers & Graphics (2006), 702-713.
3. Melchior, J., Grolaux, D., Vanderdonckt, J. and Van Roy, P. A toolkit for peer-to-peer distributed user interfaces: concepts, implementation, and applications. In *Proc*. EICS 2009, 69-78.
4. Google. The new multi-screen world: Understanding cross-platform consumer behavior. Technical report, August 2012. http://www.google.com/think/research-studies/the-new-multi-screen-world-study.html
5. Melchior, J., Grolaux, D., Vanderdonckt, J. and Van Roy, P. A model-based approach for distributed user interfaces. In *Proc*. EICS 2011, 11-20.
6. Bardram, J., Gueddana, S., Houben, S. and Nielsen, S. ReticularSpaces: activity-based computing support for physically distributed and collaborative smart spaces. In *Proc*. CHI 2012, 2845-2854.
7. Hartmann, B., Beaudouin-Lafon, M. and Mackay, W. E. HydraScope: creating multi-surface meta-applications through view synchronization and input multiplexing. In *Proc*. PerDis 2013, 43-48.
8. Hosio, S., Jurmu, M., Kukka, H., Riekki, J. and Ojala, T. Supporting distributed private and public user interfaces in urban environments. In *Proc.* HotMobile 2010, 25-30.
9. A. Peñalver, E. Lazcorreta, J. J. López, F. Botella, and J. A. Gallud. 2012. Schema driven distributed user interface generation. In *Proc*. Interacción 2012.
10. Fisher, E.R., Badam, S. K. and Elmqvist, N. Designing peer-to-peer distributed user interfaces: Case studies on building distributed applications. International Journal of Human-Computer Studies. 2013.
11. M. Manca, F. Paternò: Extending MARIA to Support Distributed User Interfaces. Distributed User Interfaces 2011: 33-40
12. L. Frosini, M. Manca, F. Paternò: A framework for the development of distributed interactive applications. EICS 2013: 249-254

[1] http://hiis.isti.cnr.it/IUDSM/index_en.html