A Framework for the Development of Distributed Interactive Applications

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

Marco Manca
HIIS Laboratory – ISTI-CNR
Via G. Moruzzi, 1
56124 Pisa (Italy)
marco.manca@isti.cnr.it
+39 050 621 3117

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 the associated software architecture to manage user interfaces that can be distributed and/or migrated 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. We also report on its current implementation, and an example application.

Author Keywords
Multi-device User Interfaces, Multi-user User Interfaces, 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 mobile devices have increased in number, and people spend more and more time using them. This has made it possible to create many environments where people spend long time interacting with various devices in sequential or in parallel [4].

In order to better exploit such technological offer often people would like to dynamically move components of their interactive applications across different devices with various interaction resources. Thus, there is a need for novel frameworks that facilitate the development of interactive applications that can be dynamically distributed across various devices.

This is an area in which some research contributions have already been proposed. Some contributions have been dedicated to investigating design spaces indicating various important relevant dimensions for this type of applications [3][6]. Such dimensions can be addressed in different ways. For example, some authors [1] proposed a solution for migrating existing Web applications, while herein we put forward a new solution for supporting distribution and migration in the development of new applications. A toolkit for peer-to-peer distribution of user interfaces was presented in [5], but it requires the use of specific libraries, while our framework can be exploited in different implementation environments. DeepShot [2] is a framework for migrating tasks across devices using mobile phone cameras, but it does not support user interface distribution across multiple devices at the same time.

In particular, our work aims to provide designers and developers with a framework that allows them to obtain applications in which the interactive components can be dynamically distributed across various devices. It allows developers to obtain applications that can have multiple instances at the same time for different groups of devices and users.

In the paper we first introduce our approach, then we describe the distribution manager and the commands that it is able to handle. We report on the underlying architecture, and the protocol used for the communication among the components. The last part is dedicated to describing the current implementation, an example application, conclusions and future work.

OUR APPROACH

We propose an environment called *Distribution Manager* composed of a client-side library for the development of Distributed User Interfaces (DUI), and run-time support for the management of the dynamic distribution. One advantage of our approach is that it does not require the use of a fixed server, as in [1], but it allows dynamic sets of devices to organize themselves in order to support the distribution. In addition, the versions of the distributed interactive applications for the various devices should not be pre-developed at design time, but can be created dynamically at run-time according to the indications defined by the developers.

The ultimate goal of the work is to provide developers with a framework to easily develop applications supporting distribution without having to implement the necessary protocol of communication and the run-time support to manage the distribution.

EICS'13, June 24–27, 2013, London, United Kingdom.

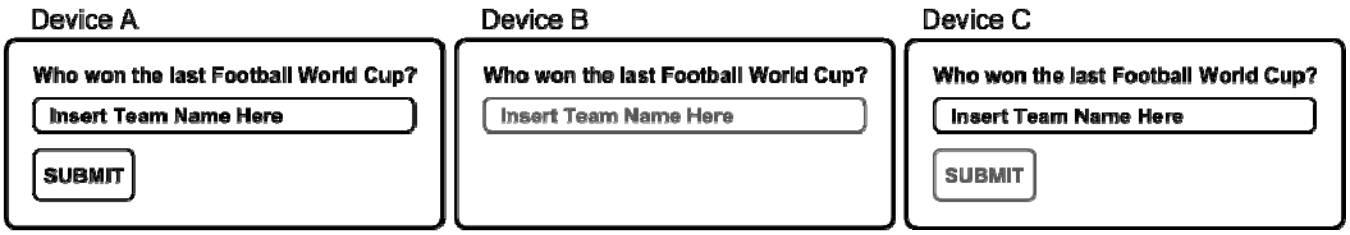

Figure 1. Example of Components Distribution.

With our framework the application can be distributed through dynamic sets of devices. The first device accessing the distribution service provides the initial distribution configuration. If later on new devices access the distribution service for the same application instance then they inherit such distribution, and can modify it if they have the appropriate access rights.

In general, it is possible to create different distribution groups composed of different devices for the same application.

THE DISTRIBUTION MANAGER

The framework is logically divided into two main components: one running on the devices where the application interactive parts are presented (we will refer to this as *client side*) and the other in a device which behaves as distribution manager (*engine side*). Using our framework the engine part does not necessarily reside on a fixed server, but when the application starts the user can configure the application to access a distribution engine running on their own mobile device or in another device. For example, in a situation where external network availability is not guaranteed, the devices can create a LAN and set one of the devices as *master,* which will act as the distribution *engine*.

The *client side* part is a library which provides application developers with facilities to:

- subscribe a device to the UI distribution service;
- request a UI distribution change to the *engine side;*
- receive notification from the *engine* of UI distribution changes for the device.

The *engine side* part provides capabilities to:

- subscribe devices to distribution changes;
- filter (depending on configurations and credentials provided by the client) and process requests for UI distribution changes;
- notify UI distribution changes to involved devices.

The requests for distribution changes can be specified by one command called **ASSIGN**. It has been designed in such a way to be easy to understand, compact, and with the possibility of obtaining flexible results. The command takes three parameters as follows:

ASSIGN(*what*, *inputEnabled*, *target*);

Where:

- ***what*** : identifies an interface part, typically the ID of the element or the container of elements that has to be distributed.
- ***inputEnabled*** : is a Boolean value. If this parameter is set to *False,* any UI events assigned to the element identified by *what* are not enabled.
- ***target*** : specifies to which device(s) the interface part identified by *what* should be distributed.

The *target* devices can be identified by lists of:

- Device Types (e.g. Mobile, Desktop), so that all the devices of the type indicated that have been subscribed to the service will receive the updates;
- Device IDs (e.g. Device 1, Device 54);
- Device Roles (e.g. Guide, Tourist, Widescreen), in this case groups of devices can be identified for the role that they play within the application.

Figure 1 shows an example in which a UI composed of one *container* and 3 internal elements: a *Label,* a *TextInput,* a *Button.* Three devices are subscribed to the distribution service (**A**, **B**, **C**). Suppose that we want to show the *container* on all devices and the other elements in the following way:

- Device **A** shows all the elements contained in *container* and all element are *enabled* to receive input events;
- Device **B** shows only *Label* and *TextInput* (but does not show the *Button*). *TextInput* receives the feedback of the entered input from other devices but the user cannot insert any value through it.
- Device **C** shows all the elements, the *Button* is visible, but is deactivated.

Such assignments are then transmitted to the relevant devices through distribution update commands defined in our protocol, each of them can contain multiple assign commands. The distribution commands requested for this configuration are the following:

- **ASSIGN**("*Container*", True, [A, B, C])
- **ASSIGN**("*Label*", True, [A,B,C])
- **ASSIGN**("*TextInput*", True, [A,C])
  ASSIGN("*TextInput*", False, B)
- **ASSIGN**("*Button*", True, A)
  ASSIGN("*Button*", False, C)

In general, a distribution command on an application element clears the previously defined assignments. Thus, for example if we want to move one object from one device to another then it is sufficient to assign it to the second device. This implicitly removes the element from the initial device. If the element needs to be redundant over multiple devices then it is sufficient to indicate such devices in the target field.

ARCHITECTURE

Figure 2 shows the proposed architecture divided into its two main components: *engine side* and *client side*. Furthermore, there is a component that represents the application that uses the *client side* library; the application logic is responsible for associating UI events with distribution change commands.

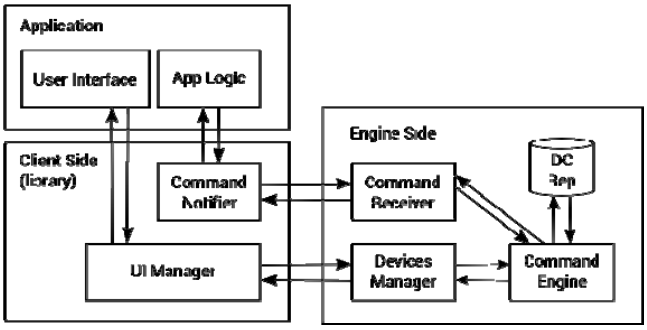

Figure 2. Overview of the Distribution Manager architecture.

Engine Side

The *engine side* part of the architecture is logically composed of three main components: **Command Receiver**, **Command Engine**, **Devices Manager**.

The **Command Receiver** is the one responsible for:

- receiving new device subscriptions to the UI distribution service;
- receiving requests of distribution changes commands;
- filtering and controlling if the received request can be made by the device sent it.

The **Command Engine** is the component responsible of taking in charge the requests of UI distribution changes (from Command Receiver), and process them to calculate the new distribution status.

The processing consists in: expanding the list(s) identified by the *target* (when the ASSIGN command *target* is a list of *Device Types* and/or *Device Roles*) to obtain the actual devices involved; checking if the ASSIGNMENT command (for *what*) is compatible with the ASSIGNMENT of the ancestor elements in the interactive application structure.

Once the new state of the distribution has been processed, it is passed to the **Devices Manager,** which is the component responsible for communicating to each subscribed device the new distribution status.

There is also a **Distribution Configuration Repository** (**DC Rep**), which is used by the Command Engine to retrieve application specific information. For example, it can provide the description of the actual distribution state, allowed roles for the application, and the needed credential to get such a role. The DC Rep is also the component responsible to maintain the status of distribution at any time (we will refer to this as *Actual Status*).

Client Side

The *client side* part is logically composed of two components: the **Command Notifier** and the **UI Manager.**

The **Command Notifier** is the component responsible to send:

- device subscription commands;
- distribution update commands;

The **UI Manager** is the component responsible for:

- receiving UI update commands;
- receiving the *Actual Status* of UI distribution when a device is subscribed. The *Actual Status* is the distribution description in a certain moment;
- perform actions to make UI update commands effective (i.e. show or hide the UI element, enable or disable event associated to elements, enable or disable input elements).

COMMUNICATION PROTOCOL

The content of the distribution communication protocol is encoded in XML. Each request from client to the engine contains an *Application ID* defined at development time. The engine uses this ID to retrieve the configuration for the specific application on DC Rep.

Some application can be session based. This means that for the same application more than one session can be created, which may be even active at the same time. For this reason each request created by the client contains a *Session ID* as well. In this way it is possible to identify the group of devices running the same application instance.

A session is newly created by the first device that subscribes with a certain *Session ID*. To create a new session a device should have the right to do it. If the new session is accepted the engine requests the initial distribution status to the client.

Sequence Diagram

Figure 3 shows a sequence diagram that represents the communication flow between clients and engine.

The diagram describes a case where Device D creates a new session, and the engine requests it to send the initial distribution state. Then, device A registers successfully to distribution and another one (C) is refused, because it did not provide acceptable credential.

After the subscription Device A receives from the *engine* the *Actual Status* of distribution and it shows the UI accordingly. Then, device A requests a distribution change. In the example the request has effect on all devices subscribed in that moment (A, D).

After this event, device B requires to be subscribed, and once the request is accepted, it asks for a new distribution change. In this case we can notice that this request has effect only on devices A and B.

We can notice that after a new request of subscription to the distribution service the *Actual Status* of distribution is sent to the subscribed device. This allows a device to subscribe itself at any time and not only at the beginning of the session.

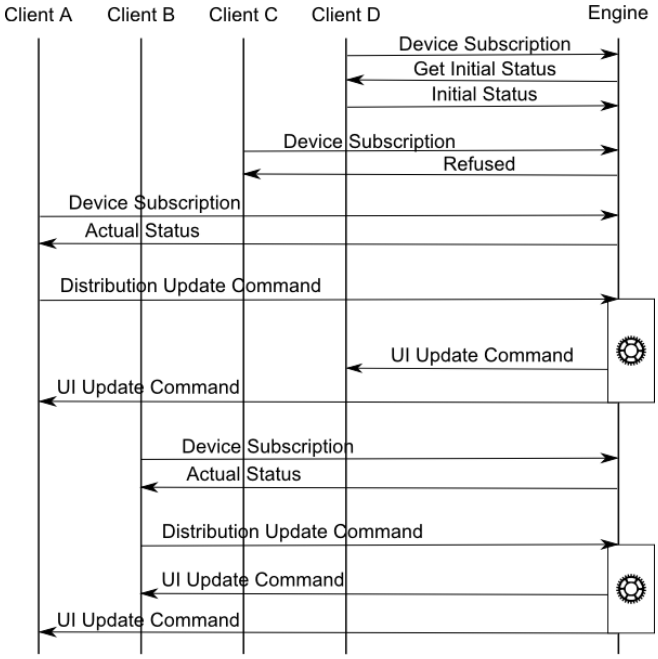

Figure 3. Sequence Diagram describing an example of interactions among multiple devices.

MESSAGE TYPE

Device Subscription

The following XML snippet shows an example *Device Subscription* Request.

```
<DeviceSubscription
    applicationID="4071d6cb-f11e-4f37-9f1d "
    sessionID="fec8e5bd-12f1-4c8b-9bee">
    <Device id="JDQ39015d2a5018500202"
        name="Nexus 7"  type="MOBILE">
        <Connectors>
            <Connector type=”HTTP”
            uri=”http://146.48.107.155:5454/” />
        </Connectors>
    </Device>
</DeviceSubscription>
```

When a device requests a subscription it communicates its own ID, a name and the device type it belongs to. Furthermore it also sends the connector type that will be used by the engine to contact the client for notification of distribution changes. The choice of the connector is made by the developer and depends on the device capabilities and the current configuration.

One example is when the application recognizes that it is running on the same device of the engine, in this case an API connector is preferred because the direct API call consumes less resources (e.g. battery, memory and CPU, bandwidth) than the HTTP connector.

In the XML example the device requesting the subscription is a Mobile device that can be contacted by the engine using HTTP at the URL provided.

On device subscription, the *engine* elaborates the new *Actual Status* of distribution for the device. The *Actual Status* is communicated as a list of *UIUpdateCommands* described below.

Distribution Update Command

When a user accesses an application developed using our framework some user-generated events can trigger requests for distribution changes in one or more devices subscribed.

Such requests are communicated to the *engine* using the **Command Notifier** through distribution update commands that can include various assignments. An example of the information sent for such requests is presented in the following XML snippet

```
<DistributionUpdateCommand
    applicationID="4071d6cb-f11e-4f37-9f1d "
    sessionID="fec8e5bd-12f1-4c8b-9bee">
    <Assign>
        <What>
            <id>Image54</id>
        </What>
        <InputEnabled>True</InputEnabled>
        <Targets>
            <Target>
                <ID>
                    JDQ39015d2a5018500202
                </ID>
```

```
            </Target>
        </Targets>
    </Assign>
    <Assign>
        <What>
            <id>Image54</id>
        </What>
        <InputEnabled>False</InputEnabled>
        <Targets>
            <Target>
                <ID>
                    5018500202KH5J99
                </ID>
            </Target>
        </Targets>
    </Assign>
</DistributionUpdateCommand>
```

The example XML shows a case where the element with ID (Image54) is distributed in different ways on two devices: in the first one the element is enabled to receive the events associated by the developer (i.e. tap or long press), in the second one the events are not enabled.

UI Update Command

When the engine receives a new distribution request, if accepted this has potential side effects on one or more devices, which are notified through update commands.

The following XML snippet shows an example of *UI Update Command*

```
<UIUpdateCommand>
    <elementID> Image54</objectID>
    <visible>True</visible>
    <inputEnabled>False</inputEnabled >
</UIUpdateCommand>
```

The snippet is received by the device identified with *5018500202KH5J99* in the previous example. The *UI Update Command* informs the device to show (visible is True) the element identified by the *Image54* ID and to disable the events associated to it.

IMPLEMENTATION

We have developed a prototype as proof-of-concept supporting the proposed approach.

Our prototype is implemented in Java and has been developed to run on Android and Desktop platforms. On both implementations the *engine* responds to requests thought a Servlet.

We have tested the library with different Android devices, which have different computational capabilities, screen resolution and size ranging from small mobile phones to large tablets, and desktop systems with screens with various sizes.

Another capability in our implementations is the API connection. The API connection is used when the *engine* and the *client* are running on the same device because it avoids access to the network and thus saves resource consumption.

In addition to the advantage that the framework does not require a fixed server, another important contribution is that the development of a client supporting the distribution requires limited effort.

Fig. 4 shows an example of code that can be used to register the device to distribution.

```
public void subscribeDevice() {
    CommandNotifier notifier = CommandNotifier.getInstance();

    ClientConnector clientConnector = new ClientConnector(master);
    EngineConnector engineConnector = new EngineConnector(master);

    Device thisDevice = new Device(Utility.getDeviceName(),
            Utility.getDeviceID(), clientConnector);

    notifier.setDevice(thisDevice);
    notifier.setEngineConnector(engineConnector);
    notifier.subscribeDevice();
}
```

Figure 4. Example of code to subscribe a device to distribution.

For each event that generates a distribution change just a few lines of code are needed to notify it to the engine as shown in Fig. 5

```
public void notifyDistributionUpdateCommand(What what,
                                boolean inputEnabled, Target target) {
    CommandNotifier notifier = CommandNotifier.getInstance();

    Assign assign = new Assign(what, inputEnabled, target);

    notifier.notify(command);
}
```

Figure 5. Example of code to notify a distribution update to the engine.

EXAMPLE APPLICATION

We have applied our framework prototype for a mobile guide. The resulting application runs on mobile devices and can be used to enrich the visit experience of tourists inside a museum.

The scenario considered aims to exploit opportunistically large screens that can be encountered during the user visit and are identified through QR codes.

The application selects the large screen acquiring the QR code and allows users to select images, video or textual information to show on the large screen to enrich the visit experience by sharing them with other visitors. Figure 6

shows the user interfaces in the two devices before (top) and after (bottom) the distribution.

Figure 6. Example of distribution involving a mobile device (smartphone) and a large screen.

In the mobile application the tap event is associated to a *DistributionUpdateCommand*. In this case, the *command* contains two ASSIGN commands: one to the sending device with *InputEnabled* to *True* and the other to the large screen (identified by its ID which is known thanks to the QR code) with *InputEnabled* to *False*. This situation corresponds to that described as an example in the Distribution Update Command paragraph.

A long press event on a resource instead results in an assignment only to the mobile device, removing it from large screen if present.

It is important to highlight that this application is not just a solution to share multimedia content but it also changes dynamically the interaction capabilities of the various parts of the application distributed across multiple devices.

CONCLUSION & FUTURE WORK

We have presented a framework for dynamic distribution of interactive components, composed of a library and a run-time support. We have also reported on the current implementation and its use for a specific application.

The main contribution of our framework is that it eases the development of applications that support UI distribution, and does not require a fixed server to support runtime distribution.

Future work will be dedicated to investigating further improvements to our solution able to make it even more flexible, optimize the battery consumption on mobile devices, and address security issues by introducing customizable security policies in the architecture presented.

We also plan to carry out a study with application developers in order to gather further empirical feedback regarding the easiness of distributed UI development and suggestions for improvements.

ACKNOWLEDGMENTS

This work is part of a project co-funded by Regione Toscana, ISTI-CNR, IIT-CNR and Softec s.p.a which aims to create a framework to develop applications able to support distributed user interfaces in mobile environments. More info at http://giove.isti.cnr.it/IUDSM/index_en.html.

We also thank Zeno Amerini (Softec s.p.a.) for useful discussions.

REFERENCES

1. Bellucci, F., Ghiani, G., Paternò, F., Santoro, C. Engineering JavaScript state persistence of web applications migrating across multiple devices, In *Proc.* ACM SIGCHI 2011, ACM Press (2011), 105-110.
2. Chang, T.H., and Li, Y. Deep Shot: A Framework for Migrating Tasks Across Devices Using Mobile Phone Cameras. In *Proc.* CHI 2011, ACM Press (2011), 2163-2172.
3. Demeure, A., Sottet, J.-S., Calvary, G., Coutaz, J., Ganneau, V., and Vanderdonckt, J. The 4C Reference Model for Distributed User Interfaces, in *Proceedings of ICAS '08*, IEEE, 2008, 61-69.
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., Van Roy, P. A toolkit for peer-to-peer distributed user interfaces: concepts, implementation, and applications, In *Proc.* ACM SIGCHI 2009, ACM Press (2009), 69-78.
6. Paternò, F., Santoro, C. A logical framework for multi-device user interfaces. In *Proc.* ACM EICS 2012, ACM Press (2012), 45-50.