Migratable User Interfaces: Beyond Migratory Interfaces

Donatien Grolaux
CETIC ASBL
Aéropole
Rue Clément Ader, 8
B-6041 Gosselies, Belgium
+32-71-91 98 08
dg@cetic.be

Peter Van Roy
Dept. of Computing Science and Engineering
Catholic University of Louvain
Place Sainte-Barbe, 2
B-1348 Louvain-la-Neuve, Belgium
+32-10-47 83 74
pvr@info.ucl.ac.be

Jean Vanderdonckt
School of Management (IAG)
Catholic University of Louvain
Place des Doyens, 1
B-1348 Louvain-la-Neuve, Belgium
+32-10-47 85 25
vanderdonckt@isys.ucl.ac.be

ABSTRACT
The migration of a user interface (UI) is the action of transferring a UI from a device to another one, for example from a desktop computer to a handheld device. A UI is said *migratable* if it has the migration ability. This paper describes how the QTk toolkit was extended to provide migratable UI and what API is provided to the developers. Basically an indirection layer has been introduced between the application and the actual representation of the UI. The migration of a UI is achieved by firstly creating a clone of the state of the site displaying the UI, secondly by changing the indirection to point to this clone. The API provides a way to specify if (the entirety of) a window can be migrated or not at construction time. A migratable window returns a universal reference that can be given to any site with whom a network connection is possible. This reference can be used by a receiver widget to migrate the window there. Interestingly, a migratable window can itself contain a receiver widget configured to display the content of another migratable window: all windows are transparently migrated along. Also a window (stationary or migratable) may contain one or more receiver widgets : it is possible to dynamically compose a UI from several different UIs.

Keywords
Attachability, migration, detachability, migratory UI, multiple computing platforms, multi-surface interaction, naturalness, plasticity.

INTRODUCTION

Remote access to applications is common place nowadays: X11 remote displays [15], VNC [16], Windows Terminal Server [17] to name a few of them. These solutions are based on an external service such that the applications are not aware of the remote access nor can they control it. Also the granularity of the remote access is very broad; in general whole screens are exported. Because of that, these mechanisms cannot provide multi-platform interactions where the user interface of an application is spread over several devices. A famous example of such interaction is the Painter's Palette problem: painters use a combination of pencils, a color palette and an uncluttered canvas, when traditional paint applications display both the color palette and the canvas on the same screen. If the application was designed to allow the migration of the color palette and the toolboxes to another device like a handheld computer, then a fully uncluttered canvas on the screen would be possible; the handheld device would be the digital equivalent to the Painter's Palette [1,4].

In other words, the multi-platform interaction can be achieved by offering the migration ability and control to the applications themselves. This paper explains how this capability is introduced at the toolkit level. First, the stationary version of the toolkit is detailed. Second, the migration extension is explained, with an example of the API. Third, the migration mechanism itself is detailed by explaining the remote access mechanism then the migration between two sites. Fourth, the conclusion details how this toolkit has been used to solve the Painter's Palette problem on a real application.

INTRODUCTION TO QTk

This work is based on the QTk toolkit [6] for the Mozart system [12]. QTk in itself is an extension of the Tk [13] module that offers an object-oriented binding to Tcl/Tk [14]. The Mozart platform implements the Oz programming language. QTk takes advantage of Oz records to introduce a partly declarative approach for programming user interfaces. Oz records are tree-like data structures that generalize XML by allowing the embedding of live language references. QTk uses correctly formatted Oz records to specify:

- The initial widgets of a window and their initial states.
- The geometry placement of these widgets and their behavior upon window resize events.
- Basic user interaction like button clicking (depends on the type of the widget).
- For each widget, a handle specifies a variable that will be bound at construction time to the object controlling that widget.

The handles fill the gap between the declarative and the imperative worlds:

- The description record specifies the initial state of the window, and some statically definable aspects of its dynamic behavior (e.g. geometry behavior upon window resize, basic actions)
- The handles are hooks over the widgets to control them at runtime. They are equivalent to the objects created by usual imperative toolkits.

```
D=td(lr(glue:we
        label(text:"Enter your name")
        entry(handle:E glue:we))
     lr(glue:we
        button(text:"Ok" action:OKPROC)
        button(text:"Cancel"
               action:CANCELPROC)))
W={QTk.build D}
{W show}
{E set("Type here")}
{E getFocus}
```

This example shows the different aspects of QTk:

- `D` is the description record
    - The `td` container widget organizes widgets top-down
    - The `lr` container widget organizes them left-right
    - The `handle` feature of the entry widget references `E`: once the window is built, `E` will be a reference to an object that interacts with this particular entry widget.
    - `OKPROC` and `CANCELPROC` are procedure names; their bodies are defined elsewhere in the program. Basically, they will close the window while taking into account (resp. discarding) the value of the entry.
- The `QTk.build` function takes a description record as parameter, builds the corresponding UI and returns an object controlling the window. As a side effect, all handles of the description are linked to objects controlling the corresponding widget. In this particular example, this is where `E` is linked to an object controlling the entry widget of the window.
- `{W show}` applies the show method to the `W` object. This instruction makes the constructed window visible to the user (by default, it is kept invisible).
- `{E set("…")}` and `{E getFocus}` respectively changes the text of the entry widget and gives the user focus to it.

MIGRATION PROCESS API

Before describing the migration mechanism itself, we first show how it is provided to QTk developers. A function `QTk.newMigratable` is introduced:

```
W={QTk.newMigratable D}
```

Where `D` is a description record accepted by `QTk.build`. This function returns an object that represents a virtual window instead of a real one. As such, this window lacks the decoration of real windows (title bar, close button, top left system menu), and does not support window specific methods like `show`. However it provides the `getRef` method that returns a unique universal reference for this window.

```
UniversalRef={W getRef($)}
```

A `receiver` widget is introduced: it defines a rectangular area of a window that can display a remote user interface when given its universal reference:

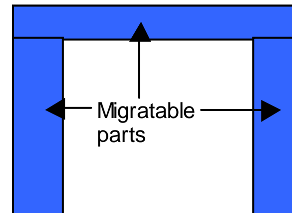
```
DW={QTk.new td(receiver(glue:nswe
                        handle:R))}
{DW show}
{R set(ref:UniversalRef)}
```

It is up to the application to pass the universal reference of a migratable window between interested processes. Using the Mozart distribution subsystem, this is done by passing around a string of characters that looks like a URL [18]. For example an application can use tcp broadcast to get this reference from processes willing to offer a migratable user interface on the same LAN.

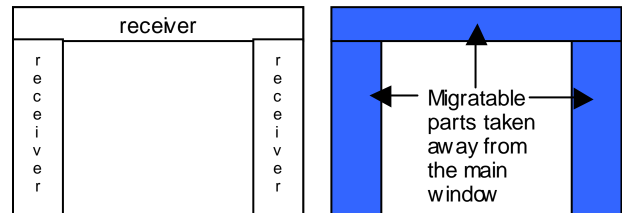
Handles of migratable windows

From the application point of view, handles of virtual windows are equivalent to handles of real windows: it's impossible to tell if a handle is from a migratable user interface or from a stationary one. As there is also no difference between description records of stationary UIs and migratable UIs, we say that the migration capability is *transparent* to the application. It is also important to notice that applications where only subparts of the UI are migratable are also easy to implement by composition:

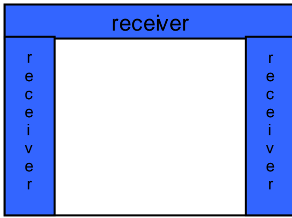
1. Split the different migratable subparts of the UI in rectangular composition of widgets



2. Take these compositions away from the description record of the window, and replace them by receiver widgets



3. For each subpart, create a migratable window from its corresponding description
4. Create the main application window
5. Place the migratable windows into the receivers that replace them: visually the output is the same as the stationary UI.



It's up to the application to decide if migratable subparts should be migrated together or not.

MIGRATION WITH QTK

This chapter details the general concepts behind this migration in a very broad view first, and then details the implementation issues.

Stationary case

Let an application A whose user interface is displayed in the single window M. M is composed of widgets M_i. To each M_i corresponds an object O_i (in the object-orientation sense) that serves as the application interface for that widget. For simplicity, let's first assume the following restrictions (they will be removed later):

[R1] Only the application can change the state of M_i, there is no interaction possible by the user.

[R2] The set of widgets contained in M is constant during its existence : if $\exists$time t where $M_i \in M$, then $\forall$time t', $M_i \in M$ also. This restriction assumes that M is populated by all its widgets at creation time. This is not the case with most graphical imperative toolkits (the UI is incrementally created by commands to add widgets), however this is the case with the QTk toolkit where a complete window is built at once from a description record.

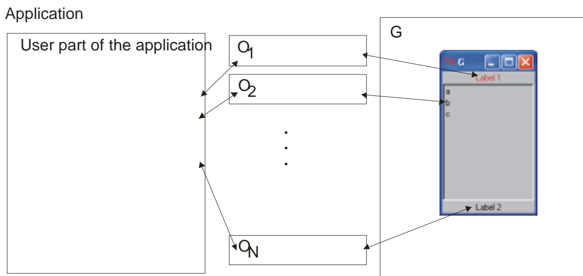


Figure 1. Stationary case

If M were purely stationary when A is running, the application would respect the scheme of figure 1.

Remote case

Before taking the migration process into account, let's first consider a **stationary remote** UI. Let's consider that M is running on a distant site[1] D different from the site L running A. The O_i objects are thorn apart between both sites :

- they should be at the D site otherwise nothing links the UI with the actual application
- to achieve transparency, they should also be at the L site : the application should be able to access these objects as in the stationary case

To achieve transparency, O_i objects are replaced by P_i proxy objects on the L site. These proxy objects are such that the application can't tell the difference between them and the original O_i objects; interactions from the application are relayed from L to D, while interactions from the user are relayed from D to L. Both sites use a communication manager to work together. The general scheme is shown in figure 2.

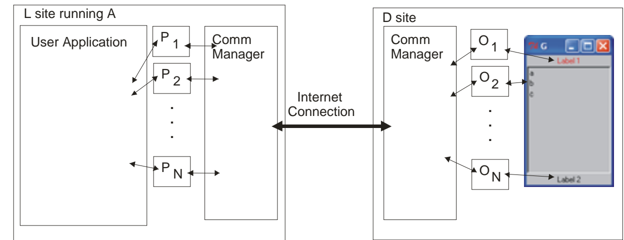
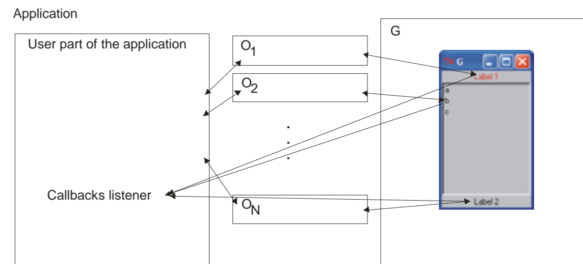


Figure 2. Remote case

For example, if the application wants to change the text of the label O_1, it would normally do so by invoking the method `set(text:"Some new text")` on `O1`. This invocation is applied to P_1 instead, which tells the communication manager (CM) of A about it. The CM of A relays this information to the CM of D which invokes this method on O_1, changing the actual widget's content. From the application's point of view, there is no difference between P_i objects and O_i objects, thus the application doesn't need to change when using a migratable UI or a stationary one. From the user's point of view, the modification of states by the application are relayed to the remote site where the UI is displayed, consequently it is the same user interface no matter where it is physically situated.

Callbacks

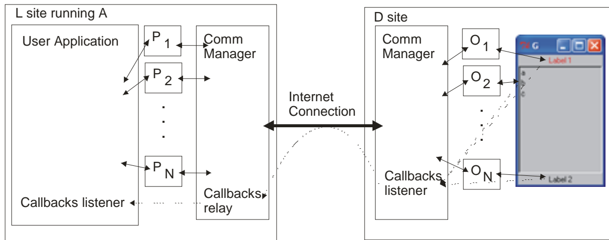
Feedback from users is generally managed by a callback system where widgets are configured to tell the application about specific events. Let's remove the [R1] restriction in the stationary case:



This is one of the possible schemes for callbacks: the application uses O_i objects to configure the widgets to react

[1] A site is a process, however to keep in mind that D and L may be running on different computers, the term site is used here.

to certain user events. When these events occur, the widgets send a message to a listener in the application itself. It's up to the listener to decide what to do in response of these messages. In the remote case, the CMs relay these callbacks to the application. The event configuration commands are sent to P_i instead of O_i. These commands are relayed to O_i objects by the CMs. They are applied at D such that the messages are sent to a listener inside the CM of D itself. Each time a message is received by this listener, it is relayed to the CM of A which in turn sends it to the listener inside the application.



Implementation of remote user interfaces with the QTk toolkit in the Oz programming language

Several properties of Oz make the implementation of such relay mechanism easy:

- Oz has a full support for object-oriented programming. The QTk toolkit provides an object-based abstraction for controlling widgets.
- Oz supports a symbolic tree data structure: records. They have the form:
  `label(feat1:val1 feat2:val2 … featN:valN)`
  where
  - `label` is an atom. An atom is any text between single quotes, or a single word starting by a lowercase character which is not a keyword of the language (for example hello, world and 'hello world' are atoms)
  - `featX` are integers or atoms. When not present, an implicit numbering is used instead (`label(val1 val2 … valN) == label(1:val1 2:val2 … N:valN)`)
  - `valX` can be any data structure of the language, including variables or records.

  Records are natural data structures of the language, with full support to manipulate them. The description mechanism of QTk is based on Oz records.
- Method invocations are based on records. If `O` is an Oz object, `{O set(bg:"black")}` calls the method `set` of `O`, with `"black"` as the `bg` parameter. What is important to notice is that `set(bg:"black")` is in fact an Oz record. If a variable is set to this record `MyRec=set(bg:"black")`, then `{O MyRec}` is equivalent to the invocation above.
- Classes support an `otherwise` method that is called by default if a method invocation uses a method name not defined in the class definition of the object. Instead of raising an exception when an undefined method is invoked on an instance of such a class, the `otherwise` method is invoked instead, with the actual record used for invocation given as parameter.
- Oz supports transparent distribution, which means that Oz data structures have a distributed protocol attached to them. For example, records are duplicated using a transparent marshalling process.

Using the `otherwise` method technique, it is easy to create P_i objects that accept any kind of method invocations, and send them to the CM of L for relay to the other site. Eventual exceptions are caught by the CM of D, and sent back to the CM of L which injects them to the application. Note that the message exchange between the two CMs benefits from the transparent distribution of Oz: there is no need to marshall the data structures into a stream of bytes for sending through the network. Event notifications are always configured by the `bind` method of O_i. The CM of D site catches these methods to configure them to use its own callbacks listener. This listener relays all messages to the CM of L which injects them to the callbacks listener of the application.

Migration process

Let's consider a site L running a migratable user interface MW displayed at the site D_1, and let's migrate this user interface to site D_2. In essence, a migration consists of these steps:

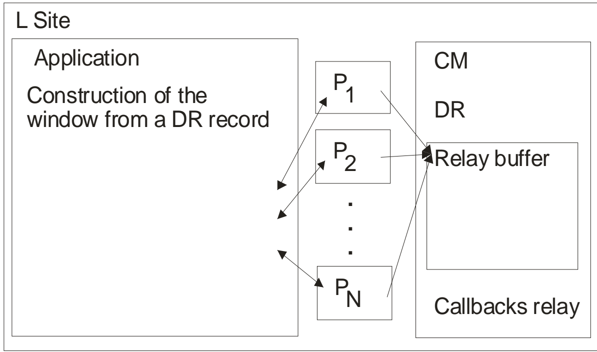
1. Isolate the migratable user interface from the application to prevent interactions during the migration process
2. Create at D_2 a UI equivalent to the one being at D_1, ie a clone.
3. Disconnect the CM of L from the CM of D_1, and connect it to the CM of D_2
4. Resume the interaction with the application

Isolating the application is necessary because the whole migration process takes some time but should be atomic wrt the application interaction. Creating a UI at D_2 equivalent to the one at D_1 consists in reproducing all its state and behavior observable by the user:

1. For all widgets MW_1^i at site D_1, create a widget MW_2^i at site D_2 of the same type and with the same geometry (same placement configuration inside the window).
2. For all widgets MW_1^i, for all visual parameters of MW_1^i, set the corresponding visual parameter of MW_2^i to the same value.
3. For all event bindings of widgets MW_1^i, set the same event bindings to MW_2^i.

Implementation of the migration using QTk and OZ

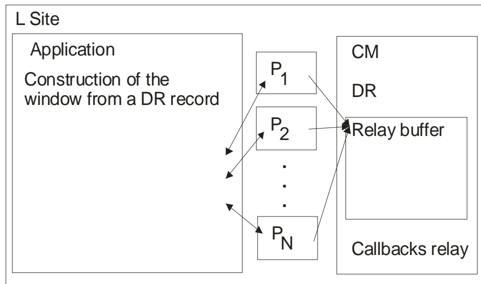
Let a migratable window built from a description record DR. The handle parameters of DR are bound to P_i objects and a CM object is created. The original DR record is also stored by CM. At this stage, there is no display site D yet.



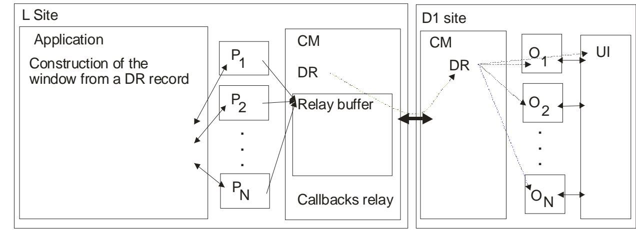
So far there are no real O_i objects, so how should the P_i objects behave if the application tries to interact with them ? At least these two different designs can be envisioned:

1. P_i objects can fully emulate O_i objects by themselves.
2. P_i objects cannot emulate O_i objects at all and they explicitly rely on them.

The first one is the most powerful, but involves a huge work of development and maintenance: for each different kind of widget, the toolkit should implement two classes with the same signature, one for the stationary case and the other for the migratory one. The second approach is far less expensive to implement as it only requires creating a polymorphic class for migratable widgets in general, no matter what their particular properties might be. QTk uses this approach. As a result, the migratable widgets cannot work until there is a display site. When not connected, method invocation messages are buffered; only when connected these messages are processed by the display site.



When the first remote display site D_1 connects to the CM of L, the DR record is sent. D_1 creates the actual user interface and the O_i's from this DR. At this moment, the application can start working with the remote user interface; the buffered messages are sent first.



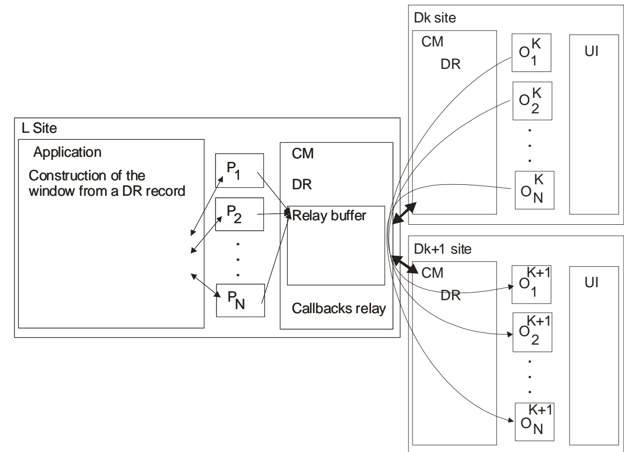
When migrating to a D_{k+1} site, the actual user interface and $O^{k+1}{}_i$'s are still created from the DR record. However the visual aspects of the widgets might have changed since their creation time, and the D_{k+1} site should reflect that. Let's define:

- VA(O)={v | v is a visually observable aspect of the widget controlled by O}
- get(O,v): returns the current value of the visual aspect v of O.
- set(O,v,s): sets the visual aspect v of O to s.

After the user interface and $O^{(k+1)}{}_i$'s are created at D_{k+1}, $\forall$ i in 1..N, $\forall$ v in VA($O^k{}_i$): set($O^{k+1}{}_i$,v,get($O^k{}_i$,v)). In practice, P_i's are used to store the visual parameters: P_i's contain a dictionary that supports the operations:

- get(P,v): returns the value of the key v of the dictionary of P
- set(P,v,s): sets the key v of the dictionary of P to s.

When disconnecting from a display site D_k, $\forall$ i in 1..N, $\forall$ v in VA($O^k{}_i$), set(P_i,v,get($O^k{}_i$,v)). When connecting to a display site D_{k+1}, $\forall$ i in 1..N, $\forall$ v in VA($O^{k+1}{}_i$), set($O^{k+1}{}_i$,v,get(P_i,v)).



Widget manipulations

The [R2] restriction assumes that the set of widgets in MW is constant over time. Under this restriction, QTk can offer migratable UIs: the set of widgets of MW is defined at creation time and remains the same over the different migrations as the same DR is used each time to set up the initial display of the different D_is. Let's remove the [R2] restriction and consider an application that:

- adds one or more widgets to MW
- deletes one or more widgets from MW
- changes the geometry management of one or more widgets of MW

Implementation of widget manipulations

From the implementation point of view, this is a difficult problem for several reasons. Before detailing them, it is important to understand how QTk manages this kind of functionality. The declarative approach of the construction record does not by itself support the dynamic manipulation of widgets in the user interface. Consequently, QTk provides several dedicated container widgets for solving this problem. The simplest one is the placeholder widget that reserves a rectangular area of the UI:

```
placeholder(handle:P)
```

The P handle object can be used to display any new widget as long as the placeholder exists. The creation of widgets is however still based on a declaration record. For example, to create a label inside the above placeholder, the application could do:

```
{P set(label(text:"Hello world"))}
```

and to replace the label by a button:

```
{P set(button(text:"Button"))}
```

as the placeholder can contain only one widget at a time, the previous command replaces what was currently displayed (the label) by the new widget (the button). Note that the new widget can be a container widget with a complex configuration:

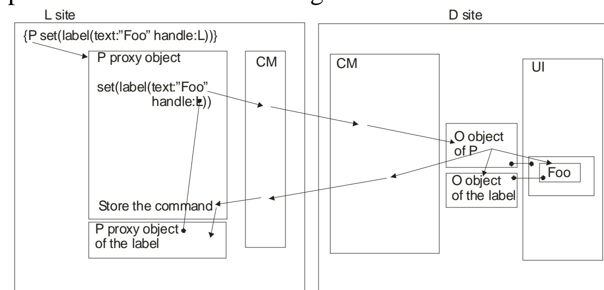
```
{P set(td(lr(glue:we
      label(text:"Type your name here")
      entry(handle:E glue:we bg:white))
   lr(glue:we
      button(text:"Ok"     handle:BOk)
      button(text:"Cancel"
             handle:BCancel))))}
```

The implementation problems related to the migration are:

- The proxy objects are general objects that cannot make the difference between a method invocation that creates a new widget and other method invocations that don't.
- Only the display site can detect a method invocation that leads to the creation of a new widget.
- When migrating, dynamically created widgets have to be created again on the new site. The only way to do it correctly is to replay the method invocation that created the widget at first.

When a method invocation is received by P_i, it is relayed to the D site through the CMs as usual. When applied by the O_i object, the new widget creation is detected: it is created along with the handle objects as usual and the CM of D is notified of the creation of these objects. This notification is relayed to the CM of A which creates proxy objects for all the newly created objects, and connects them to the handle parameters in the description record used to create them. Also, the method invocation used to create these new widgets is stored in P_i.

When migrating, right after the window is built from the original DR record, for all proxy objects, if they contain method invocations to create widgets, replay these methods, but instead of creating new proxy objects for each newly created widget, use the corresponding already existing proxy object. The rest of the migration process stays the same: for all widgets in MW, restore their visual parameters and event bindings.



Fault tolerance and performance issues

The default migration configuration assumes a fault-free environment. This is a reasonable assumption when prototyping an application, or in those special cases where the environment is guaranteed to satisfy it. For normal Internet situations, this is not enough, and there should be a way to manage fault tolerance. According to the migration protocol, at most one display site can be connected to the application site at any time. Both sites can go down:

- If the application site goes down, there is no running application anymore, so there should be no UI too. In such a case, the receiver widget of the display site is emptied by destroying whatever it contained.
- If the display site goes down, an assumption of the migration protocol is broken: the P_i don't have the opportunity to store the visual states of the O_i before disconnecting from the display site. However QTk can still use the last successful migration data to restore the UI in that state.

The fault-tolerance consists in restoring the visual aspects of the widgets in case a display site goes down. One scheme would be to store this information each time it is available. For example if the user application changes the color of a button to red, it has to issue a method invocation of the form: {ButtonHandle set(background: red)}. This information could be caught by the proxy object and stored somewhere. However it involves that the proxy objects have enough knowledge of the semantic of the widget they pretend to be to differentiate between method invocations that change visual aspects of the widgets and others that don't. As mentioned before, this approach would require a big development and maintenance effort. Also if the user changes the state of a widget (typing a letter in an entry field for example), this information

has to be brought to the application site; that causes a network overhead, and again the proxy objects should have enough knowledge of the widgets they pretend to be to know what to do of this information.

Another approach is used: let the application define the way the storage and update of the visual aspect of the widgets are done. The application can define them by classes of widgets and/or for specific instances of widgets and/or let the default behavior. The default behavior is the one described in the migration protocol above:

When disconnecting from a display site D_k, $\forall$ `i` in `1..N`, $\forall$ `v` in $VA(O^k_i)$, $set(P_i, v, get(O^k_i, v))$.

When connecting to a display site D_{k+1}, $\forall$ `i` in `1..N`, $\forall$ `v` in $VA(O^{k+1}_i)$, $set(O^{k+1}_i, v, get(P_i, v))$.

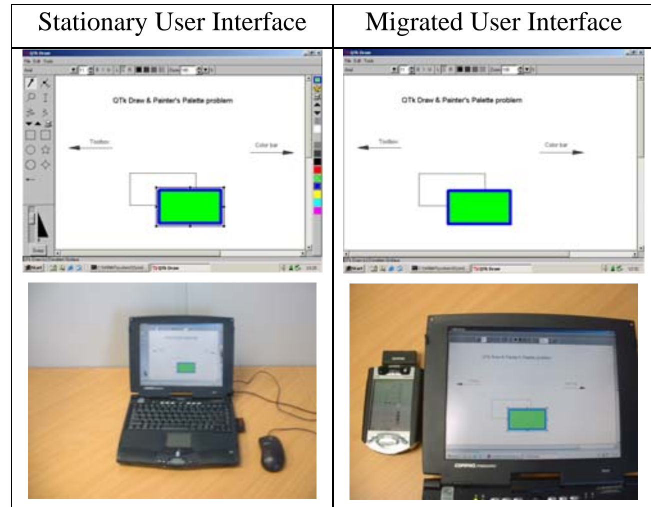
The idea is that one can make a prototype application with migratable widgets at nearly no cost assuming a poor behavior in case of failures. When making it fault-tolerant, the developer has to redefine these aspects so that no action at all is taken when disconnecting from a display site, and when connecting to a site, the visual aspects are restored from the inner state of the application. As a result, the migration process is fastened: the restoration of visual aspects depends only on the changes the application expects, and is fault tolerant as the application restores these parameters in a coherent way with its inner state. On real examples, the speedup of the network operations has often been observed to be more than 50%.

Painter's Palette problem

To highlight the benefit of transparent migration, we modified a drawing application written for QTk *before* migratable UIs existed to solve the Painter's Palette problem. In the real world, a painter uses an uncluttered drawing area, its tools and colors are on a separate palette so they don't get in the way of the drawing. On the contrary, computer's drawing applications use the screen to represent both: the drawing area is reduced by the place taken to display the different toolboxes. QTkDraw was modified so that the left and right toolboxes of the window are now migratable. A user can send them for example to an external iPaq handheld: this is the digital version of the Painter's Palette [1,4]. The modifications in the original code were:

- extract the toolboxes part of the description record of the window, and replace them by receiver widgets
- create migratable windows from the extracted definitions
- create the main application window using the modified description
- place the migratable windows in the corresponding replacement receivers At this stage, there is no visual difference between the original version of the application and the migratable one.
- offer the universal references to the outer world by using a tcp socket.

A small palette application was created: it builds a window with two receivers, connect to the socket of the main application to get the universal references back, and injects them to the receivers widgets. All these modifications required changing and writing less than 40 lines of code for an application of more than 8000 lines of code.



Also if new functionalities are included into the application, they can be implemented as if the user interface was purely stationary. In a sense, QTk provides a migratable capability to user interfaces, that is decided at the window's construction time. Because of the way windows are first constructed, it is a straightforward and low-cost process to take advantage of.

RELATED WORK

The first steps that have been made towards migratory UIs where those virtual window managers capable of remotely access an application over the network. Some representative examples include: X11 remote displays [15], VNC [16], Windows Terminal Server [17]. These solutions are based on a service external to the user's applications. This type of control is under the responsibility of the underlying operating system. Therefore, an interactive application cannot control its own migration. Pioneering work in migration has been done by Bharat & Cardelli [3]: their migratory applications are able to move from one platform to another one at run-time, provided that the operating system remains the same. While this is probably the first truly migrating applications, the main restriction is that it requires to migrate the *whole* application from one point to another, including the UI. It is the same case when an application should be transferred to another user [5]. In [2], only the UI is migrated from one computing platform to another, all being web-based. At run-time, the user can choose the platform where to migrate.

In this paper, migratable UIs go one step further: they can migrate partially or totally, from one platform to many others, even with different operating systems. At run-time, they can be decomposed and recomposed when needed.

Recent studies clearly demonstrate the need and the opportunity of having user interfaces distributed across several displays, both theoretically [4] and empirically [7,10,11]. The notion of several displays has been expanded into the notion of multi-interaction surfaces [4,8], beautifully exemplified in the Pick & Drop interaction technique [9].

CONCLUSION

This paper presents a graphical toolkit supporting migratable user interfaces. From the application point of view, this is a transparent process : there is no difference between using a stationary UI, and a migratable one. A drawing application has been changed to behave like the painter's palette : the toolbox and the color selection bar can be taken away from the main window, and migrated to any other computer like a handheld PDA for example. The difference between the stationary version of the application and the migratable one is around 20 lines of code out of more than 8000. The application that receives the migrated UI is also around 20 lines of code. Note that the core of the application can be extended as if the whole application was purely stationary. As a window can contain an arbitrary number of migrated UIs at the same time, it is also possible to dynamically compose a UI from different migrated components. One could imagine several different applications managing different aspects of a unique problem : their UIs are conveniently migrated to a single place. For example, an application could dynamically probe the wealth of a system (free ram space, free disk space, CPU occupation), and be run in several instances on several computers. The system administrator migrates the UIs from all these applications into a single window. This window is migrated between his office computer when (s)he is in front of his desk, and his or her laptop computer when (s)he is away. Also the development cost of this application is almost the same as the development cost of a stationary diagnostic tool, very little change is required to make the UI migrate. In summary, this toolkit provides low cost migration mechanism that enables us to have more freedom with multi-platform ubiquitous user interfaces.

Acknowledgment

This work was funded at CETIC (www.cetic.be) by the Walloon Region (DGTRE) and the E.U. (ERDF and ESF).

REFERENCES

1. Ayatsuka, Y., Matsushita, N., Rekimoto, J. HyperPalette: a Hybrid Computing Environment for Small Computing evices, in Proc. CHI'2000 Extended Abstracts (The Hague, April 1-6, 2000), ACM Press, 133-134.
2. Bandelloni, R., Paternò, F., Platform Awareness in Dynamic Web User Interfaces Migration, in Proceedings of Mobile-HCI2003.
3. Bharat, K.A., Cardelli, L. Migratory Applications Distributed User Interfaces, in Proc. of UIST'95 (Pittsburgh PA, November 14-17, 1995), ACM Press, 133-142.
4. Coutaz, J., Lachenal, C., Calvary, G., Thevenin, D., Software Architecture Adaptivity for Multisurface Interaction and Plasticity, in Proc. of IFIP WG2.7 Workshop on Software Architecture Requirements for CSCW–CSCW'2000 Workshop (Philadelphia, December 2-6, 2000), ACM Press, New York, 2000. Accessible at http://iihm.imag.fr/coutaz/ifipcscw2000/work shop.html
5. Dewan, P., Choudhary, R., Coupling the User Interfaces of a Multiuser Program. ACM Transactions on Computer-Human Interaction 2, 1, 1-39.
6. Grolaux, D., Van Roy, P., Vanderdonckt, J. QTk: A Mixed Model-Based Approach to Designing Executable User Interfaces, in Proc. of 8th IFIP Working Conference on Engineering for Human-Computer Interaction EHCI'01 (Toronto, May 11-13, 2001), Chapman & Hall, London
7. Grudin, J. (2001). Partitioning digital worlds: focal and peripheral awareness in multiple monitor use, CHI'2001, 458 - 465
8. Rekimoto, J., Masanori, S. Augmented Surfaces: A Spatially Continuous Work Space for Hybrid Computing Environments, in Proc. of CHI'99 (Pittsburgh, May 15-20, 1999), ACM Press, New York, 378-385.
9. Rekimoto, J. Pick-and-Drop: A Direct Manipulation Technique for Multiple Computer Environments, in Proc. of UIST'97 (Banff, Alberta, Canada, October 14-17, 1997), ACM Press, 31-39.
10. Strope, J., Putting Usage-Centered Design to Work: Clinical Applications, in Proc. of 1st Int. Conf. on Usage-Centered, Task-Centered, and Performance-Centered Design foruSE'2002, Constantine, L. (Ed.), Ampersand Press, Rowley, 2002.
11. Tan, D.S., Czerwinski, M. (2003). Effects of Visual Separation and Physical Discontinuities when Distributing Information across Multiple Displays. INTERACT 2003 Ninth IFIP International Conference on Human-Computer Interaction, Zurich, Switzerland.
12. The Mozart Programming System, accessible at http://www.mozart-oz.org/
13. Window Programming in Mozart, accessible at http://www.mozart-oz.org/documentation/wp/index.html
14. Tool Command Language, accessible at http://www.tcl.tk/
15. The X11 Consortium, accessible at http://www.x.org/
16. Virtual Network Computing, accessible at http://www.uk.research.att.com/vnc/
17. Windows Terminal Server, accessible at http://www.microsoft.com/windows2000/technologies/terminal/default.asp
18. Distributed Programming in Mozart - A Tutorial Introduction, chapter 3: Basic Operations and Examples, accessible at http://www.mozart-oz.org/documentation/dstutorial/node3.html#chapter.examples