Dynamic Distributed User Interfaces: Supporting Mobile Interaction Spaces.

Geert Vanderhulst

2004 – 2005

Abstract

As the possibilities and popularity of mobile computing devices increases, there is an opportunity to accomplish more and more desktop-related tasks with them. Most tasks require a Graphical User Interface (GUI) that allows users to tap buttons, make selections, fill in forms, etc. Because of the diversity in mobile devices, it is difficult to develop a single GUI that runs on all of these devices. One of the difficulties is the lack of a common platform and/or GUI toolkit. The heterogeneity of platforms and toolkits can be masked by using markup languages to describe an interface on a high level. The use of these description languages is one of the core ideas in this dissertation. Another concern is the adaptation of the GUI to the context of use. Our goals are to clarify what adaptation of an interface involves and why it is important.

Since mobile devices have limited screen space and varying input methods, it may be appropriate to split up a GUI and distribute parts to several devices. This poses new challenges such as an efficient distribution mechanism and wireless communication techniques to let the distributed parts talk with each other and the application logic. We present a framework that is able to generate and distribute GUI descriptions for custom applications, that can be rendered on a wide variety of (mobile) devices. The emphasis is on open standards, platform/toolkit/language independence and a light-weight framework.

Summary (Dutch)

In deze verhandeling bestuderen we hoe een grafische user interface (GUI) kan gedistribueerd worden over verschillende apparaten en we stellen hierbij een eigen framework voor. In het bijzonder richten we onze aandacht op mobiele toestellen zoals PDAs en smartphones. Mobiliteit wordt alsmaar belangrijker en het gebruik van zakcomputers neemt toe. Het spreekt echter voor zich dat een interface voor een PDA aan andere eisen moet voldoen dan een interface voor een desktop PC. Bij de ontwikkeling van een GUI voor zakcomputers moet er rekening gehouden worden met de beperkte schermgrootte en liefst ook met de beschikbare invoermogelijkheden (toetsenbord, touchscreen, drukgevoelige pen, ...) of andere factoren. Het is belangrijk dat het distributieproces rekening houdt met de verscheidenheid aan toestellen in de omgeving.

Een eerste vaststelling is dat de traditionele ontwikkeling van een GUI, namelijk rechtstreeks gebruik makende van een grafische toolkit zoals Swing of Qt, weinig geschikt is voor gedistribueerde interfaces. De hoofdreden hiervoor is dat grafische toolkits over het algemeen platform en programmeertaal afhankelijk zijn. Een oplossing voor dit probleem bestaat erin de GUI te *beschrijven* op een hoog niveau in plaats van deze meteen in programmeercode te implementeren. Deze beschrijving dient dan op het device zelf gevisualiseerd te worden, met behulp van een zogenaamde 'renderer' die mogelijk gebruik maakt van een beschikbare toolkit. De beschrijving zelf is porteerbaar naar verschillende toestellen. We maken hierbij een onderscheid tussen web interfaces (XHTML, CSS, JavaScript, ...) en specifieke User Interface Description Languages (UIDLs). Hoewel web interfaces slechts een beperkte widget set ondersteunen, kunnen ze op een grote ondersteuning rekenen. De meeste hedendaagse systemen hebben een web browser aan boord die web interfaces kan visualiseren. (XML-gebaseerde) UIDLs hebben dan weer het voordeel dat ze krachtigere mechanismen ondersteunen op grafisch gebied, maar daarentegen moeilijker te renderen zijn op verschillende toestellen. Het is een gegeven dat een UIDL zonder beschikbare renderers weinig praktisch nut heeft, hoe goed de specificatie ook moge zijn.

Eigenlijk zou het niet mogen uitmaken voor het besturen van een applicatie of een apparaat een web interface presenteert aan de gebruiker, dan wel bijvoorbeeld een UIQ gestyleerde interface. Daarom beschouwen we als kern van een distribueerbare GUI een *willekeurige* beschrijving in XML (XHTML, UIML, ...). Een GUI is distribueerbaar als deze in zijn geheel of in op voorhand gedefineerde delen kan migreren naar meerdere computers. Een belangrijke vaststelling hierbij is dat gebruikers vaak slechts behoefte hebben aan afgelijnde

diensten die een applicatie aanbiedt. Als een gebruiker bijvoorbeeld wat achtergrondmuziek wil opzetten via haar/zijn PDA, heeft deze normaal geen behoefte aan een playlist, maar play/stop/... knoppen zullen voldoende zijn. Het volstaat dan om slechts een *deel* van een GUI te distribueren naar het toestel van de gebruiker en optimaal gebruik te maken van de beschikbare schermruimte. Om een dergelijke 'slimme' distributie mogelijk te maken, gebruiken we een XML-schema taal (RELAX NG) om de structuur, beperkingen en types van een aantal diensten (delen van een GUI) te beschrijven. Op die manier kunnen we eender welke XML-gebaseerde beschrijvingstaal gebruiken en op gelijke manier behandelen.

We maken een onderscheid tussen user-driven, system-driven en continuous distributie. Het verschil tussen user-driven en system-driven distributie ligt vooral in de actor – de gebruiker of het systeem – die beslist welke diensten (delen van de GUI) naar welke toestellen worden gedistribueerd. Indien het systeem de actor is, is het belangrijk dat eigenschappen van toestellen in de omgeving in rekening gebracht worden om een optimale distributie te bepalen. Bovendien kan de inbreng van de gebruiker belangrijk zijn om het distributieproces enigszins te begeleiden. Verschillende profielen met informatie over diensten, gebruikers en toestellen spelen dan ook een cruciale rol bij system driven-distributie. Verder kan het wenselijk zijn om apparaten in de omgeving automatisch te detecteren, hetgeen mogelijk is met een discovery protocol zoals bijvoorbeeld UPnP. Continuous distributie bestaat erin om een GUI automatisch te herdistribueren als clients tot de interactieruimte toetreden of deze verlaten. We moeten echter uitkijken dat de interface niet al te vaak en/of drastisch wijzigt om de bruikbaarheid te blijven behouden, maar dit is een onderwerp voor verder onderzoek.

In een gedistribueerde omgeving is het belangrijk dat apparaten met elkaar kunnen communiceren. Vooral draadloos ethernet en netwerken gebaseerd op het IP-protocol zijn geschikt om hosts onderling te verbinden. Eén van de voordelen van IP is de directe link met het Internet, waardoor toestellen – naast in een lokaal netwerk – ook overal met elkaar kunnen communiceren waar een Internet-verbinding beschikbaar is (hotspots, GPRS, UMTS, ...). Een interessant protocol dat bovenop IP/TCP opereert is HTTP, vaak gebruikt in combinatie met het web of gerelateerde diensten. HTTP wordt ook vaak gehanteerd in *middleware* systemen om data uit te wisselen, en meer bepaald RPC-gebaseerde middleware. Een groot voordeel van middleware is dat het over het algemeen de heterogeniteit van apparaten, netwerken, besturingssystemen, etc. maskeert en bovendien neemt het heel wat werk uit handen van de ontwikkelaar. XML-RPC, SOAP en REST zijn vaak gebruikte RPC-mechanismen in op diensten gerichte omgevingen, en vooral de REST 'visie' valt op omwille van de eenvoud.

In het voorgestelde framework maken we gebruik van de RELAX NG schemataal om GUIs te genereren en distribueren. De kern van het framework bestaat uit de Interface Distribution Daemon (IDD) waarin een light-weight web server zit ingebouwd. De IDD biedt een platformonafhankelijke REST API aan die ontwikkelaars kunnen gebruiken om clients/applicaties aan te melden, actions/events door te sturen, etc. Eén van de problemen die we hierbij zijn tegengekomen is het feit dat het gebruikte HTTP protocol gericht is op client-server communicatie, terwijl ons communicatiemodel verwacht dat de server (IDD) ook de client kan contacteren. De introducie van een extra HTTP methode (LISTEN) blijkt

hiervoor een eenvoudige, maar afdoende oplossing.

We hebben het ontwikkelde systeem toegepast om een audio player (XMMS) en image viewer (JShow) te bedienen met behulp van een gedistribueerde interface, gespreid over verschillende toestellen (laptop, PDA, ...). Het framework is zo gebouwd dat ontwikkelaars met een minimum aan moeite *bestaande* applicaties kunnen inpluggen op het systeem en voorzien van een gedistribueerde interface. Aangezien er enkel gebruik gemaakt wordt van open standaarden, is er geen nood om een framework-eigen taal te leren en bovendien verloopt de communicatie tussen gedistribueerde GUI delen onderling en de applicatielogica geheel transparant voor de ontwikkelaar.

Acknowledgments

In the first place I would like to thank my promotor prof. Karin Coninx, dr. Kris Luyten and Chris Vandervelpen for guiding me through my thesis, but also for giving me the opportunity to cooperate with a research paper related with the subject of my thesis.

Special thanks go to my girlfriend Elke Moyens, for always being there and for the valuable time spent together, and to my family for supporting my studies and providing me with the necessary equipment.

I also want to express my gratitude to all students and roommates who made life easier and more fun. Among them are Bert Vangoidsenhoven, roommate and friend who improved life at the student apartment and campus despite his weird eating habits in the morning; Jan Bollen, our first class driver and team member in many school projects; and Koen Vanlaer, an old friend I regularly went out with.

Contents

List of Figures

List of Tables

Part I

Research Foundations

Chapter 1

Developing Graphical User Interfaces for Mobile Devices

1.1 Introduction

Technology evolves and in the mobile computer market this results in more powerful devices with large high resolution screens. These devices give rise to more and more advanced Graphical User Interfaces (GUIs) to interact with programs or just navigate through menus. The 'mobile interface' aspect also introduces new opportunities. Handheld devices can be used to control remote applications and appliances or operate together with PCs [21]. An interesting hypothetical application to take a look at is MANNA [10], since it reveals many of the challenges posed by UI development for mobile computing.

Many features rely on the portability of a GUI. A single GUI should run on a wide range of devices. However, few UI developers have the skills to build interfaces for different not to say all platforms. And even if they have the skills, it would take a lot of time to develop cross-platform GUIs by conventional and commonly used techniques. We discuss major approaches to design portable GUIs and focus on issues and requirements.

1.2 Definition of a 'good' User Interface

Two important properties of a UI are its usability and its appearance. An interface should be usable, e.g. have an intuitive placement of controls, clear buttons, obvious menu entries and so on. Besides, it should be attractive to work with. Of course, different users have different ideas and preferences of what a UI should look like. No wonder thus that many applications are customizable nowadays. Common options to personalize a UI are the adaptation of toolbars (i.e. add or remove buttons), selection of different layouts and skin support.

The usability of a UI is closely related with the device on which it is used. Since UIs are a means to allow human-computer interaction (HCI), the way on which the interaction takes

place is crucial. Interaction depends on the device's input method(s). Common input devices are keyboards, keypads, touchpads, touchscreens + pen, mice, ... Handhelds often have one or the other, but a combination like both a pointing device (e.g. mouse) and a keyboard (e.g. desktops or laptops) is rather seldom. Therefore UIs should be developed with the available input methods in mind. A GUI for keypad-only cell phones should not offer drag and drop support or expose a drawing canvas to the user, as both rely on a pointing device. GUIs with a lot of double-click actions or popup menus (triggered by a right mouse click) should be avoided on touchscreen devices. Many issues can be thought of, and several usability studies and guidelines are found in HCI courses.

Interface Consistency

The benefit of consistency is that users can leverage knowledge from previous experiences with similar interfaces. UIs can be made consistent with:

- each other; e.g. a portable UI rendered on a desktop should at least be recognizable when it is rendered on a handheld device.
- other (default) interfaces on the platform or device; e.g. Gnome-/KDE-like interfaces, Symbian Series60/UIQ style, Pocket PC look and feel, ...

If a user is used to a certain GUI running on Pocket PC, she/he should also be able to work in short time with a smartphone representation of that GUI. So, both representations should not differ too much. Controls should look familiar to apply the experience acquired by using one of the GUIs. On the other hand, it is a good thing if both GUIs share the look and feel of traditional Pocket PC and smartphone GUIs respectively. This will help users to apply device knowledge.

1.3 User Interface Toolkits

A common way for creating a Graphical User Interface (GUI) is by means of a programming language and a GUI toolkit. Such toolkits offer a high-level API to the programmer which removes the hassle of drawing individual pixels and lines. Therefore, GUI toolkits facilitate the creation of graphical interfaces. There is no need for the developer to know about low-level system calls.

The choice of a GUI toolkit is closely related with the language in which the program behind the interface is written. For instance, Java programmers will probably use the Swing or AWT API, while C++ programmers can choose between Qt, GTK, ... Toolkits like Swing or Qt (and many others) offer a complete application framework to the developer. They offer methods to catch mouse clicks, key presses and many other user actions. Unlike these, there is a lot of libraries providing only the core of a GUI toolkit: widgets. Widgets are specific

parts of a GUI such as buttons, labels, text fields, ... Two core properties of GUI toolkits include:

- **Portability**: How 'portable' is the toolkit, i.e. on what operating systems and devices can it be used?
- **Supported widgets**: What are the widgets offered by the toolkit and what are their properties?

As Java is known as a cross-platform language, one would suggest its GUI toolkits are highly portable. This is true as far as we consider the major *desktop* operating systems. Swing and AWT run on Windows, Mac and Unix based systems. However, when it comes to mobile operating systems like Palm OS, Symbian OS, Windows Mobile versions, ... things are different. In the best case one can force Swing or AWT to work using third party software. PDAs and smartphones have slower processors and less memory than their desktop variants. Therefore it figures that trimmed down or specific toolkits are being used on these devices. When mobile phone companies claim that their devices support Java application/games, they actually mean that their devices have J2ME class libraries on board. J2ME stands for "Java 2 Platform, Micro Edition" and is – as the name says – a tiny micro edition with a light-weight Virtual Machine (KVM), compared to the standard or enterprise edition for desktop systems.

Because of the performance gap between handhelds and desktops and especially their different screen sizes, GUI toolkits are more or less developed for a particular range of (similar) devices. Also among PDAs and smartphones there is a diversity of used toolkits. For instance, Symbian phones run different versions of the Symbian OS with different GUI toolkits (known as Series 60, 80, 90, UIQ, ... phones). [30] explains how to design portable UIs *just* for the different Symbian OS flavours. So if it is even not trivial to develop portable UIs within successive versions of a mobile OS (using its built-in GUI toolkit), it will definitely be hard to extend their portability to other mobile platforms.

Toolkits can also be (partially) ported to other operating systems – some actually are – but this involves a lot of work and they should be maintained. For instance, if the 'original' toolkit gets updated with new features or fixes, the ports should be updated too. Therefore it is probably not the best idea to pick a GUI toolkit and assume it will be supported on all or most devices. Neither it would be useful, since a (full) Swing to cell phone port would turn the phone in an overloaded and slow device. The other way round, a handheld GUI toolkit to desktop port would result in a feature-less toolkit, compared to other desktop UI toolkits.

A better approach is to *describe* the interface on an abstract and toolkit-independent manner. The description could then be rendered – in theory – using *any* toolkit. There is, however, a big issue: not all GUI toolkits support the same set of widgets. Thereby it can be difficult to describe an interface in terms of widgets and still require that it can be rendered on a wide range of devices (using different GUI toolkits). The latter is also related with the used *renderer*. A renderer is used to transform the description to the actual interface.

1.4 Markup Languages

GUIs are usually put down in a *programming* language. The program code should then be compiled into (native) machine code. Advantages of this approach are:

- Fast rendering, since machine code is directly understood by the computer (low-level) or by a Virtual Machine (mid-level).
- Easy integration with applications (i.e. written in the same programming language).

However, major disadvantages are:

- Limited portability: depends on the portability of the programming language as well as on the portability of the used GUI toolkit.
- The slightest change in the GUI code (program code) requires recompilation.

Apart from programming languages, markup languages can be used to *describe* an interface. Markup languages are self-describing what makes them both readable to humans and machines. Their strengths are high degrees of portability and the fact that they need not to be compiled in order to be understood. What they do need, however, is an *interpreter* to read and process data. For example, browsers interpret HTML (HyperText Markup Language) code and render it to a web page *on the fly*.

Most markup languages are based on the XML (eXtensible Markup Language) [34] syntax. XML is standardized by the W3C. It is easy to parse and there are tons of (free) tools to read, write, transform, ... XML files on almost every platform (also the mobile ones).

1.4.1 User Interface Description Languages

A User Interface Description Language (UIDL) defines syntax and semantics to describe a user interface. A UIDL should be declarative so that it can be edited by hand, but it should also be formal in order to be understood and analyzed by software.

UIDLs can offer different levels of abstraction. On the one hand an abstract high-level description has the advantage that a UI can be described independently of a widget set what makes it highly portable. On the other hand, a more concrete UI description (e.g. toolkit-specific) is easier to render and does not suffer from the fact that GUI toolkits offer varying widget sets. It applies, in general, that the more abstract a UI description is, the more complex its renderer(s) will be.

A common technique used in many UIDLs is the mapping of Abstract Interface Objects (AIOs) to Concrete Interface Objects (CIOs) [31, 27]. A "range indicator" can for example be mapped on a slider or spinner widget. This mapping is a step towards adaptable UIs, i.e. dynamically decide what mapping is the most appropriate based on certain constraints.

More on this in section 1.6. In addition, the CIO can be mapped on a specific toolkit, e.g. a device's default toolkit. This way, UIs can be rendered that are consistent with the ones from (default) applications running on the device.

[26] gives an overview of the most significant XML-compliant UIDLs and compares them on different areas. These UIDLs include UIML, AUIML, XIML, Seescoa XML, Teresa XML, WSXL, XUL, XISL, AAIML, TADEUS XML. Although some of them are still in development phase, it seems there is plenty of choice. Non of the mentioned UIDLs should be considered as *the* UIDL. All have their pros and cons, mainly dictated by the goals for which they are intended. For instance, XIML has a very high expressivity, but it (currently) lacks advanced tool support. UIML on the other hand is one of the most restrictive UIDLs, but it is the most supported by software [26]. [26] also endorses the claim that the real attractiveness of a UIDL heavily depends on its tool support: it is meaningless to possess a refined specification of a UI that cannot be rendered or only partially.

UIML

UIML (User Interface Markup Language) [1] has two benefits compared with the other UIDLs: it is being standardized by OASIS and it is rather supported by software. UIML is also independent of a widget set and it claims to be device-independent as well [18]. A renderer either interprets and renders UIML directly on the client device or compiles it to another language (like WML, HTML, program code). Since both approaches generate a UI *automatically*, without user intervention, it is important that the resulting UI is useful and visually appealing. Especially for devices with a small screen this is important. For example, a button may be renderered too small to fit its label. Or the button may be rendered too big so that other widgets are 'repressed' by it. As these and many other possible issues depend on the renderer, the designer (or should we say 'describer'?) of the UIML files can do few if anything to avoid them.

The renderer plays a pivotal role, so let us take a look at the currently available renderers for UIML. There already are quite a few and they roughly fall apart in two categories: the ones provided by Harmonia (commercial) and those contributed by individuals (mostly open source). These include renderers for Java, C++, .NET, Symbian, ... A smartphone running Symbian will require another renderer than a Pocket PC with .NET support. In this case we have a single device-independent UI description, but multiple device-dependent renderers. In fact, 'device-dependent' renderer is not completely true, as a .NET renderer for instance is portable to different platforms. Unfortunately, it is not portable to *all* platforms and neither are the other renderers. It is simply not possible at the moment to write such an all-platforms supporting renderer, because of the diversity in (mobile) operating systems and their lack of a common Virtual Machine or API. Therefore it is important that *different* renderers perform a good job with the *same* UIML file(s) as input.

1.4.2 Web Interfaces

We already mentioned HTML as a markup language, but we did not yet link it with UI development. (X)HTML[1] is *the* language to layout web pages. It differs from the UIDLs in the way that XHTML does not describe web pages *only* in terms of widgets or in an abstract manner. On contrary, it specifies concretely what the page will look like and how data is presented: in a table with two columns, in italic or bold, what colour, ... Yet, XHTML supports 'Forms'. XHTML Forms provide a modest set of widgets like buttons, checkboxes, radio buttons, ... Note that these Forms (widgets) are just part of the HTML specification, while widgets are the core elements of any 'real' GUI toolkit. If a web page offers controls to the user to input/change data or execute scripts or (small) applications on the server, we speak of a Web Interface (WI).

WIs are everywhere. There are thousands of examples on the web. Take for example an online book shop (e.g. `http://www.amazon.com`). The web page is in the first place a frontend for a database. One can search for books on author and title and select the book(s) she/he wants to buy. Via a simple form one can provide the server with address and payment information. After confirmation, a book has been ordered through a web interface. Though most WIs reside somewhere on the web, they are also commonly used in embedded devices like routers, print servers, surveillance cameras and so on. Their use in embedded devices is advantageous because:

- The device can be controlled through a web browser; no specific software needs to be installed.
- HTML (most embedded devices still use HTML instead of XHTML) is widely supported.
- Scripts (e.g. JavaScript) can be integrated in the HTML code and provide the WI with extra functionality.
- Cascading Style Sheets (CSS) can be used to style the WI.

1.4.2.1 Mobile Web Browsers

Since web browsers have found their way to mobile devices, a router can be controlled with a PDA or cell phone *without* specific software. The only requirement is a decent web browser to render the WI. Nowadays, mobile internet is a hot topic on the feature list of mobile devices. Most devices have an integrated, pre-installed browser. Besides, a lot of third-party browsers (free and commercial) are available for the leading platforms. Browsers from Opera (also well-known on the desktop platforms) and NetFront (by ACCESS) are particularly interesting. These companies offer different versions of their browsers with the same functionality for various mobile platforms. For example, the Opera mobile browser runs on Symbian Series 60,

[1] Roughly spoken, XHTML is an XML-compliant version of the HTML 4 specification. Their descriptive power is more or less the same. More information can be found at [42].

Series 80, Series 90, UIQ, EZX, Brew, μltron, Smartphone, Qtopia and Psion devices. This means that if a WI can be rendered properly on a Series 60 Phone, it can also be rendered on a Psion device. NetFront also runs on Pocket PC and others, and ACCES provided the core technology for the Palm OS 5 Browser (based on NetFront v3.0).

Opera and NetFront support recent standards of XHTML [42], CSS [33], DOM [44], ECMAScript [16], ... Furthermore they offer special rendering techniques to cope with the small screens of mobile devices. Technologies like Smart-Screen Rendering (Opera) and Smart-Fit Rendering (NetFront) reformat a web page to fit inside the screen width and eliminate the need for horizontal scrolling.

The availability of mobile browsers with Opera and NetFront as two multi-platform solutions, is highly beneficial. It gives developers the opportunity to design a WI and test it with Opera/NetFront. The quality and support of the WI can more or less be assured *without* having to test it on different platforms or devices. Remark that this is – at least at the moment – a fundamental difference with UIDL renderers. A UIDL renderer for Pocket PC may render a satisfiable UI from a given UI description, while a Symbian renderer may screw things up. The developer cannot assure the quality of the UI on various devices without extensive tests on *all* devices.

An advantage of browsers in general is the fact that they are 'tolerable'. For instance, a non-CSS enabled browser will still render a CSS-based web page and just ignore the style sheets.

Because browsers do not use specific platform-dependent toolkits to render a web page, web interfaces do not look like the UIs of other applications running on the platform (unless they are 'skinned' to look so). WIs rather offer consistent looks on different devices.

1.4.2.2 XForms

Although web forms are still widely used, the technology is showing its age. Forms have limited features what makes even the most common tasks dependent on scripting. Other issues are poor integration with XML and no separation of the *purpose* from the *presentation* of a form. XForms [39] overcome these issues and provide updated and new UI controls as well as many other improvements.

In time, XForms should replace the current HTML Forms. Unfortunately it will take months or even years to finalize the XForms standard and get it fully implemented by the majority of web browsers. Anyway, XForms blow a fresh wind along WIs.

1.4.2.3 Local versus Remote Interfaces

There is a difference between:

- an interface for controlling a remote appliance or application, and

- an interface for controlling an application running on the same device.

WIs are an attractive option for the former sort of interface. In the latter case, however, it has little sense to develop a WI for the application. This would require to start the application and a web browser separately on a single device or use a 'browser widget' to render the WI. The separate web browser way will face the developer with complex communication issues (UI - application core) that could easily be avoided here. A 'browser widget' is neither a good idea, since it is GUI toolkit specific, rather rare and it may have limited support for recent standards (XHTML, CSS, ...). In this case, it is much easier to develop an interface using a specific GUI toolkit or describe a UI in a UIDL and invoke a renderer from within the application's code.

1.5 Hand-crafted versus automatically generated interfaces

Concrete designed UIs have the advantage that they look exactly like the designer intends them to look. The designer has full control to adapt the interface, change the arrangement of controls, ... possibly based on feedback of users. This way, the quality of the UI can be assured. On the other side, multiple concrete UIs must be provided in order to support different platforms. These may for instance include a Swing and/or Qt UI for desktop systems, Series 60/UIQ for mobile phones, ... A lot of work thus needs to be done by the designer. Also, if the UI specifications change (e.g. extra controls are requested), *all* UIs should be updated separately. Another drawback of this approach is the fact that those statically designed UIs are not very flexible. In the best case, the UI can be considered flexible if the underlying UI toolkit or framework is flexible in the sense that widgets can be added/removed dynamically or adapted to the user's preferences. Again, this kind of flexibility is toolkit and thus (in general) platform *dependent*.

On contrary, a lot of research has been done on generating UIs automatically from an abstract description. Many model-based approaches have been proposed in the past [10, 27]. A UI model is built from different UI aspects: task information, user knowledge, presentation environment, ... Model-based interface systems usually provide design tools to hide the syntax of the modelling languages and provide a convenient interface to specify the often large quantities of information that are stored in the UI model (specification).

Present projects that aim to provide (semi-)automatically generated UIs for appliances include Dygimes [8], Pebbles[2] (i.e. the Personal Universal Controller (PUC) [23]) and SUPPLE [12]. A common idea in these projects is the use of abstract (XML-based) UI specifications to define what functionality the interface should expose to the user. The presentation of those features is left to a renderer. Typically, the renderer uses a constraint-based layout management algorithm to present the UI on a certain display. Different types of constraints can be incorporated. For example, spatial constraints can define facts like "this button should always appear right to this text field". Besides, devices have a natural constraint: their screen

[2] `http://www.pebbles.hcii.cmu.edu/`

resolution. Several techniques are used to take these and other constraints in consideration to layout the UI. Two approaches are the use of decision trees [23] and turning the layout problem into an optimization problem [12].

Note that the job of the UI designer consists in the first place of designing a *functional* UI specification. In addition, she/he should possibly provide the renderer(s) with extra information (e.g. platform-dependent CIO information [8]). If the designer wants to upgrade the UI with new features, it is enough to update the functional UI specification. Unfortunately, this ease of maintenance has a serious cost: the designer loses a lot of control since the renderer decides *at runtime* what the UI will look like. But, *the* advantage with this approach is its flexibility. The UI renderer can make dynamic decisions based on environmental factors, device constraints or user preferences.

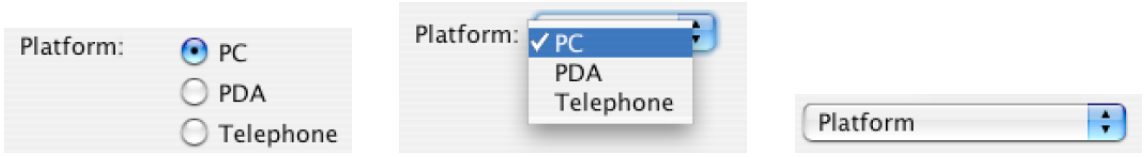
1.6 Adapatable User Interfaces

If a UI can be rendered or changed on the fly based on certain parameters or user actions, it is called 'adaptable'. Adaptable UIs are also referred to as plastic or context-aware UIs. A lot of research on this area has been collected by the CAMELEON project[3]. Most automated UI generation tools give attention to context-awareness (i.e. constraint-based layout management) [10, 27, 8, 12, 23].

Adaptability of a UI can be interpreted in many ways. The next subsections summarize some of the most important interpretations.

1.6.1 Adaptation to different devices

Different devices have different screen sizes and features and thus different UI requirements as explained in section 1.2. UIs can adapt to different screen sizes (resolutions) by reorganizing and adapting individual widgets. [6] identifies a new generation of widgets: the *comets* (COntext of use Mouldable widgETs). As a simple example, [6] shows how a set of radio buttons can shrink into a combo box (figure 1.1). Note that the combo boxes require less space than the radio buttons, while they still support the same task.

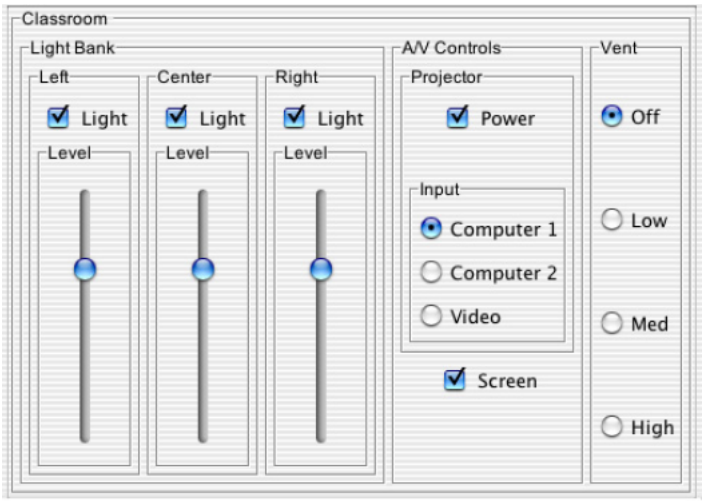


(a) Label and radio buttons (b) Label and combo box (c) Combo box incorporating the label

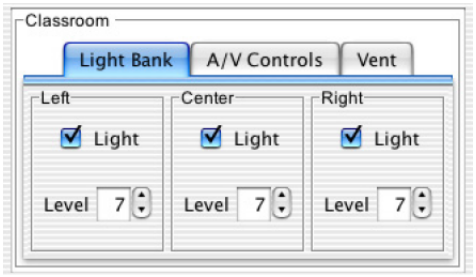
Figure 1.1: Different widgets supporting the same task, extracted from [6].

[3] http://giove.cnuce.cnr.it/cameleon.html

Similar ideas can be found in SUPPLE [12], where slider widgets are found equivalent with spinners. Figure 1.2 shows two UIs with the same functionality, but with a different placement of the controls and different widgets. The first UI is intended for large screen devices, while the second UI is more useful on devices with a smaller screen. The second UI is also better suited for devices with a touchscreen than the first. Sliders are of little sense for touchscreen devices, since 'dragging' requires a pointer. Spinner widgets are a better choice here.



(a)



(b)

Figure 1.2: Interface rendered for two devices with different size, extracted from [12].

1.6.2 Adaptation to user preferences

It is very important that end-users are satisfied with the rendered UIs and that they can work efficiently with them. Therefore users should be able to adapt the UI to their preferences, and re-arrange for instance some controls or tune their look and feel. Remark that this personal adaptation could introduce conflicts with the device adaptation. Consider for example FlexClock: a clock that expands or shrinks its UI when the user resizes the window, presented in [6]. Figure 1.3 shows two states of the UI. These screenshots show that the calendar shows up when the window is considered 'large enough'. However, users could prefer that the calendar

is *always* there and that scrollbars appear if the window is shrinked. This example stipulates that the renderer should somehow decide what adaptation is more important. We believe that user preferences should always have the highest priority.

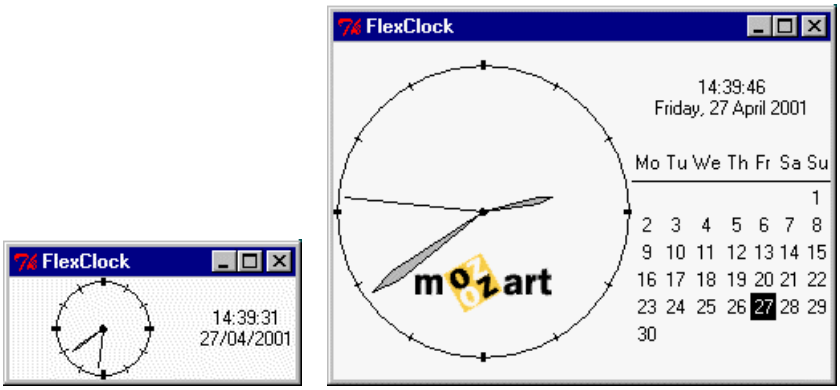


Figure 1.3: FlexClock, extracted from [6].

1.6.3 Adaptation to environmental factors

A UI may for instance have controls to switch on/off and dim lights and controls to switch on/off a projector and control its brightness/contrast. Assume the UI is used on a PDA to control the lights and projector in the room in which the user resides. Since not every room will have a projector, it has no sense to display the projector controls when the user is in a room without projector.

This interpretation of adaptability is rather complex because it depends on *realtime* information from sensors. In our lights and projector example, the lights and projector(s) in different rooms must notify their existence to the PDA. Or – more realistic – an external PC should track the position of the user and inform the PDA of the available lights/projectors.

1.7 Migratable User Interfaces

Portable and context-aware user interfaces are a big step towards ubiquitous computing. However, the adaptability of a UI has its limits. Even the smartest adaptation algorithms will turn into problems when they should adapt a heavy desktop PC interface for use on a handheld device. Take for example the interface of drawing progams like the Gimp or Photoshop. It is nearly impossible to adapt their interface to a *usable* tiny handheld interface that still provides *the same functionality*.

A way to deal with the small screen size and overcome the adaptation limitations is by migrating *parts* of an interface to one or more devices. The user then disposes of a distributed and possibly mobile version of (parts of) the initial interface. The distribution of specific UI parts poses new challenges:

- Parts should be selected carefully to make sure they can be visualised on different devices, possibly with varying screen sizes.
- Parts should be able to communicate with the main application and vice versa.
- The application state should be kept consistent among the different devices.
- ...

Notice that migratable UIs do *not* remove the need for adaptation: UI parts should still adapt to different devices!

We will elaborate on migratable UIs in part II, where we propose a framework to support them.

1.8 Summary

We discussed how markup languages and toolkits can go hand in hand and offer a viable option for cross-platform UI development. We mentioned description languages for UIs and UIML in particular, and contrasted UIDLs and their renderers with web interfaces. Thereby the importance of maintained and cross-platform software support was stated. Next, we focused on some differences between manually designed and automatically generated interfaces from the developer's point of view. Different tools aim to generate UIs automatically for different platforms and most give attention to context-awareness. We explained some interpretations of context-awareness and saw how GUIs can adapt to devices, user preferences and external factors. We also introduced migratable UIs which are the core subject of part II.

Chapter 2

Distributed Communication Techniques

2.1 Introduction

So far, we looked at how user interfaces can be rendered on different devices. However, if an interface is separated from the application logic (e.g. application and interface run on different devices), both parties should be able to communicate with each other. There are many ways to setup a communication channel; they rely on communication protocols from different layers. We will focus on wireless communication, since we primarily consider mobile (handheld) devices to render an interface and control remote applications running anywhere. This chapter also explores specific communication techniques for web browsers, because they are an attractive option for rendering interfaces on embedded devices. Section 2.5 is entirely devoted to them.

2.2 Wireless Communication

Computer networks are a rather complex object of study. The term "protocol" is often used in different contexts which does not make the whole story less complicated. Basically, a network interconnects different devices and allows them to exchange *bits*. A single bit corresponds to a 0 or 1. All data in a computer system is processed and stored in bits (at the lowest level). Low-level protocols like Bluetooth, WiFi, ... vary in the way they transfer those bits. The used strategy and hardware dictate the maximum bitrate, range and other properties of wireless networks that may differ a lot. Consequently, different protocols are used for different purposes and network types.

However, we are not much concerned about the details of these low-level protocols, but rather in their compatibility with common used high-level protocols. Most low-level protocols can deal with the Internet Protocol (IP) on which the whole Internet is based. The

IP protocol is often used in conjunction with the Transmission Control Protocol (TCP), a reliable connection-oriented protocol that allows to transfer a bytestream from one machine to another on the Internet. IP-based networks are of particular interest because they provide interoperability between different network types. For example, a mobile phone or PDA connected to the Internet over GPRS (General Packet Radio Service) can establish a connection with a laptop in the other end of the world that is also on the Internet. It does not matter whether this laptop is part of a WLAN (Wireless Local Area Network) or if it is directly connected to the Internet using a dial-up connection, as long as it is visible on the Internet. Hosts on an IP-based network (i.e. the Internet) have a unique IP address[1] by which they can be accessed.

Nowadays, mobile devices often have wireless ethernet, Bluetooth or a combination of the two on board. The Bluetooth protocol stack, however, differs quite a lot from the ethernet-based protocols like 802.11a, 802.11b and 802.11g. For instance, Bluetooth is not IP-based what makes it difficult to connect a Bluetooth Personal Area Network (PAN) to other networks. Bluetooth is rather used for synchronizing a mobile phone with a PC or exchanging some pictures with other phones (in a close range). It is also often used to connect peripherals such as keyboards, headsets, ... to computer devices without wires. An issue with Bluetooth is that the protocol defines different profiles, but devices are free to implement and offer only a subset of them. An important profile for developers to setup a connection between two Bluetooth devices is the Serial Port Profile (SPP), also addressed as the RFCOMM layer. However, experience learns that e.g. recent Nokia phones lack this profile and leave the developer in the cold. A (current) advantage of Bluetooth compared to wireless ethernet, is its lower cost and power consumption.

On contrary, devices with WiFi support (or derivatives) can integrate seamlessly in a WLAN. The WLAN may or may not be connected to the Internet and/or other networks. Moreover more and more wireless hotspots can be found in public places, allowing PDAs, laptops, ... to connect to the Internet. And if no hotspot is around, there is still an opportunity to access the Internet over GPRS, UMTS (Universal Mobile Telecommunications System) or any other technology that is available, at least if the device supports it.

2.3 Client-Server and Peer-To-Peer Model

The client-server model (figure 2.1(a)) is the most widely used. It differentiates between client and server computers in the network. A server offers one or more services by running a dedicated process that continuously listens for incoming requests and responds to them. These services can be anything: a web server delivers web sites, while a database server allows clients to query/manage databases. Clients run software that can connect and interact with a server. A typical client program is a web browser.

[1] A dynamic IP address is usually assigned by an Internet Service Provider (ISP) or local router, using the Dynamic Host Configuration Protocol (DHCP).

The other communication model between computers is peer-to-peer (P2P) (figure 2.1(b)), where application services do not reside on a central server, but peers directly communicate and exchange information between each other. To enable that, peers combine client and basic server functionality. P2P is e.g. used in the vast majority of file sharing tools that allow to download and upload files from and to peers in the network at the same time.

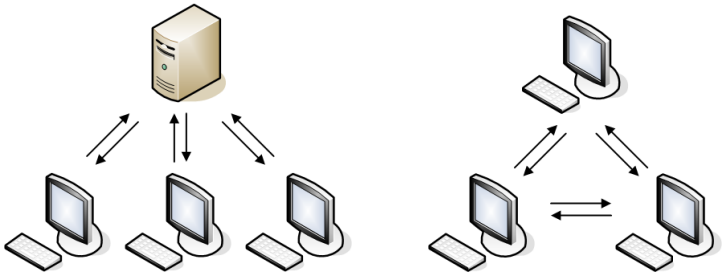


Figure 2.1: Client-server (a) and peer-to-peer (b) model.

2.4 The HyperText Transfer Protocol (HTTP)

HTTP is a request/response protocol between clients and servers. An HTTP client, such as a web browser, typically initiates a request by establishing a TCP/IP connection to a particular port on a remote host (port 80 by default). An HTTP server listening on that port waits for the client to send a request string, such as `GET / HTTP/1.1` (which would request the default page of that web server), followed by an email-like MIME message which has a number of informational header strings that describe aspects of the request, followed by an optional body of arbitrary data. Some headers are optional, while others (such as Host) are required by the HTTP 1.1 protocol [15]. Upon receiving the request string (and message, if any), the server sends back a response string, such as `HTTP/1.1 200 OK`, and a message of its own, the body of which is perhaps the requested file, an error message, or some other information. Table 2.1 and 2.2 show a sample request and response message.

Request-Line	`GET / HTTP/1.1`
Request Header Fields	`Host: www.google.com` `...`
Request Body	*empty*

Table 2.1: HTTP Request Message

The following request methods are defined in the HTTP 1.1 specification: `GET`, `POST`, `PUT`, `DELETE`, `HEAD`, `TRACE` and `CONNECT`. We refer to [15] for a detailed explanation of each method.

Status-Line	`HTTP/1.1 200 OK`

Response Header Fields	`Content-Length: 3059`
	`Content-Type: text/html`
	`...`

Response Body	`<html>`
	`  <head>...</head>`
	`  <body>...</body>`
	`</html>`

Table 2.2: HTTP Response Message

The specification document also comprises an overview of possible response status codes (e.g. the 200 in the status line of table 2.2) and their meaning.

HTTP is one of the most important protocols on the web. Web browsers/servers use it to transfer files, i.e. web pages. Furthermore, many other protocols are based on it. They rely on the request/response mechanism to exchange data, e.g. attached as body of an HTTP message. Some of these protocols are described further in this chapter.

Despite the fact that HTTP is a stateless protocol, it can deal with *cookies*, an HTTP State Management Mechanism [14, 22].

Cookies

Cookies are used to maintain the state of an HTTP session. They can store any arbitrary information the server chooses, much like global variables (without specific type) in a computer program. A cookie is set by the server in a response message using the `Set-Cookie` header field. A client (web browser) then stores the cookie in its cache. It is sent back to the server in a `Cookie` header field each time the client fires a request. However, cookies have an optional path attribute. If a path is set, the cookie is only sent along with a request if the URI of the request matches the path. Other attributes allow to specify domain, expiration date, secure flag, ... and can be looked up in the original Netscape draft specification [22] or in the newer RFC 2109 specification [14].

Cookies can provide a shopping application running on a server with the ability to keep track of the currently selected items of a client. They also allow a site to store per-user preferences on the client, and have the client supply those preferences every time that site is connected to. Furthermore cookies are often applied to identify a client (in particular a computer/browser combination), e.g. the server generates a unique id for each client and stores this in a cookie sent to the clients. A sample cookie that stores information about a preferred language is shown below.

```
Set-Cookie: language=nl;
```

2.5 The XMLHttpRequest object

The XMLHttpRequest object[2] allows web browsers to execute HTTP transactions in the background. This allows browsers to dynamically update the content of a page (HTML elements) without reloading it. The object can handle HTTP transactions synchronously (blocking) or asynchronously (non-blocking). It can be invoked from within JavaScript code and can retrieve and submit XML data directly. Received XML documents can be read and examined via the Document Object Model (DOM) interface. Figure 2.2 lists a JavaScript code fragment that illustrates how to create an XMLHttpRequest object, open an asynchronous connection and assign an event handler to it. The listing in figure 2.3 shows a skeletal event handler function that allows processing of the response content only if all conditions are right. The link in the footnote is a good source for additional information (methods, properties, ...) on the object and features an extra example in which XML data is read from RSS feeds.

```
var req;

function loadXMLDoc(url) {
  // try native object
  if (window.XMLHttpRequest) {
    req = new XMLHttpRequest();
  }
  // try ActiveX object
  else if (window.ActiveXObject) {
    req = new ActiveXObject("Microsoft.XMLHTTP");
  }
  if (req) {
    // open asynchronous connection
    req.open("GET", url, true);
    // specify event handler
    req.onreadystatechange = processReqChange;
    // submit the request
    req.send(null);
  }
}
```

Figure 2.2: Use of the XMLHttpRequest object.

Similar functionality is covered in a proposed W3C standard, Document Object Model (DOM) Level 3 Load and Save Specification [45]. In the meantime, growing support for the XMLHttpRequest object means that it has become a de facto standard that will likely be

[2] http://developer.apple.com/internet/webcontent/xmlhttpreq.html

```
function processReqChange() {
  // check if req shows "loaded"
  if (req.readyState == 4) {
    // check if the status is "OK"
    if (req.status == 200) {
      // DOM tree representation of the data returned from the server
      var xmlDoc = req.responseXML;
      // ...processing statements go here...
    }
    else {
      alert("Error:\n" + req.statusText);
    }
  }
}
```

Figure 2.3: Skeleton of an XMLHttpRequest event handler function.

supported even after the W3C specification becomes final and starts being implemented in released browsers (whenever that might be).

Microsoft first implemented the object in Internet Explorer 5 for Windows as an ActiveX object. Engineers on the Mozilla project implemented a compatible native version for Mozilla 1.0 (and Netscape 7). Apple has done the same starting with Safari 1.2. Also the Opera browser has native support for XMLHttpRequest in version 8 for the desktop platform and hopefully the mobile versions for embedded devices will follow soon. It also works with Pocket Internet Explorer (Windows Mobile 2003), but unfortunately that browser offers only partial JavaScript support what makes it difficult to update HTML code dynamically.

The XMLHttpRequest object is currently being used by Google services such as Gmail[3] and Google Maps[4], two famous sites that collect tons of hits each day. Obviously, browser developers are eager to support these sites and thereby are more or less forced to implement XMLHttpRequest (partially).

In recent blog posts[5] there is quite some commotion about AJAX (Asynchronous JavaScript + XML). AJAX is no technology on its own, but it incorporates the use of several technologies such as HTML, CSS, JavaScript, XML, DOM, ... and in particular the XMLHttpRequest object. The AJAX web application model is very promising and likely to replace the traditional web application model in time. The idea behind AJAX is to present a web interface that communicates with the server in the background; user interaction with the application happens asynchronously and updates to the interface are performed dynamically. So the user is never staring at a blank browser window and an hourglass icon, waiting around for the server to do something. With AJAX, it becomes possible to develop web applications

[3] http://gmail.google.com/
[4] http://maps.google.com/
[5] http://adaptivepath.com/publications/essays/archives/000385.php

with the same responsiveness and richness as desktop applications.

2.6 Middleware

Middleware is a class of software technologies designed to help manage the complexity and heterogeneity inherent in distributed systems. It is defined as a layer of software above the operating system but below the application program that provides a common programming abstraction across a distributed system, as shown in figure 2.4. In doing so, it provides a higher-level building block for programmers than for instance sockets that are provided by the operating system. This significantly reduces the burden on application programmers by relieving them of low-level implementation details. It rather allows developers to write simpler and portable code and build distributed applications that can interact with other applications running on various platforms.

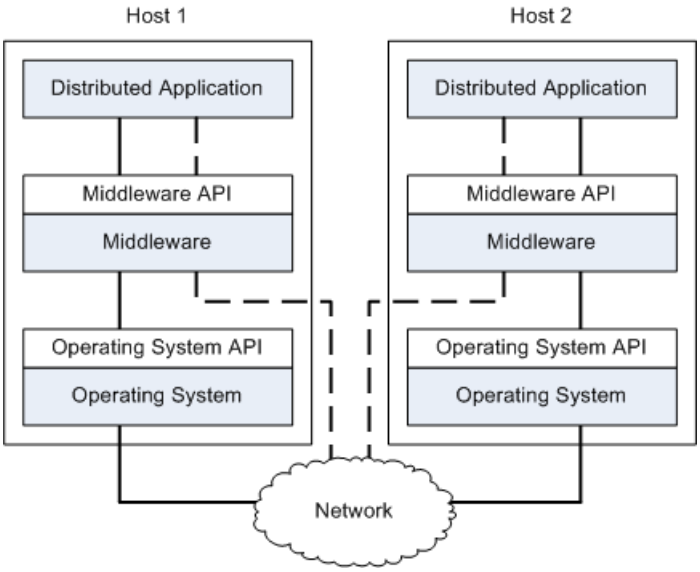


Figure 2.4: Middleware layer.

Middleware frameworks are designed to mask some of the kinds of heterogeneity that programmers of distributed systems must deal with. They always mask the heterogeneity of networks and hardware and most frameworks also mask heterogeneity of operating systems or programming languages, or both. The dotted line in figure 2.4 shows a virtual path from the first host to the second. To the developer, only the middleware API counts to maintain communication between the hosts. The underlying operating system is hidden, although the actual communication path (indicated by the solid line) passes through the OS. The middleware actually calls specific operating system functions to put data on the network and get data from it, all transparant to the developer.

There are a small number of different kinds of middleware that vary in terms of the programming abstraction they provide and the kind of heterogeneity they provide beyond net-

work and hardware. Three kinds are commonly used: middleware based on Remote Procedure Calls (RPC), Message Oriented Middleware (MOM) and middleware based on Distributed Objects. We will take a closer look at the first and the last.

2.6.1 Remote Procedure Calls

Remote Procedure Calls (RPC) is a powerful technique for constructing distributed, client-server based applications. It is based on extending the notion of conventional, or local procedure calling, so that the called procedure need not exist in the same address space as the calling procedure. The two processes may be on the same system, or they may be on different systems with a network connecting them.

A remote procedure call is analogous to a function call. Like a function call, arguments can be passed to the remote procedure and the caller waits for a response to be returned. The flow of activity that takes place during a synchronous RPC call between two networked systems is as follows. The client makes a procedure call that sends a request to the server and waits. The client is blocked from processing until either a reply is received, or the request times out. When the request arrives, the server calls a dispatch routine that performs the requested service, and sends the reply to the client. After the RPC call is completed, the client program continues.

Remote procedures are one of core technologies behind *web services*. The term 'web services' refers to a number of standards that allow web-based applications to interact with each other through open protocols, languages and well-defined Application Programming Interfaces (APIs). Examples are Google, Amazon and eBay that evolved from simple web sites to Service-Oriented Architectures (SOA). They expose an API (publically available), enabling developers to write custom front-ends or programs to cooperate with them. Because all communication is in XML, web services are not tied to an operating system or programming language. A Java application can talk with a Perl service and vice versa.

Three methods are often used to exchange data: XML-RPC, SOAP and REST. However, REST is different from the other two as it rather is a 'vision' on the standard web than a specific protocol. XML-RPC, SOAP and REST rely on the HTTP protocol (by default) to exchange XML documents. It is not unlikely that a web-based application running on a server offers an API for all three protocols. Flickr[6], for example, supports them all and in the next sections we illustrate the use of XML-RPC, SOAP and REST with the services API of this photo sharing site.

2.6.1.1 XML-RPC

XML-RPC[7] is designed to be as simple as possible, while allowing complex data structures to be transmitted, processed and returned. There are many implementations available that

[6] `http://www.flickr.com/services/api/`
[7] `http://www.xmlrpc.com`

span a wide range of operating systems and programming languages. Figure 2.5 shows an XML-RPC request and response message.

2.6.1.2 SOAP

SOAP (Simple Object Access Protocol) [37] can be considered as a powerful extension of XML-RPC. It is an XML-based protocol that consists of three parts: an envelope that defines a framework for describing what is in a message and how to process it, a set of encoding rules for expressing instances of application-defined datatypes, and a convention for representing remote procedure calls and responses. SOAP 1.2 is a W3C Recommendation, but is often criticized for being too complex and overequipped for what should be a simple task. Figure 2.6 shows a SOAP request and response message.

2.6.1.3 REST

REST (REpresentative State Transfer) is different. In the REST approach, a procedure call is made by a traditional HTTP request where the URI contains the name of the procedure and its arguments. For REST, no sophisticated technology is needed: a default web server is sufficient. It is clean and simple, as shown in figure 2.7.

2.6.2 Distributed Objects

Distributed object middleware provides the abstraction of an object that is remote yet whose methods can be invoked just like those of an object in the same address space as the caller. Distributed objects make all the software engineering benefits of object-oriented techniques such as encapsulation, inheritance, and polymorphism available to the distributed application developer. Objects are marshalled by the client, transferred over the network, and unmarshalled by the server transparant to the developer. This process is handled by stubs (proxies). Three commonly used distributed object middleware frameworks are CORBA, Microsoft's DCOM/COM+ infrastructure and Java RMI. We will only introduce them shortly and avoid details, because that would lead us too far.

2.6.2.1 CORBA

The Common Object Request Broker Architecture (CORBA) is a standard for distributed object computing. It is part of the Object Management Architecture (OMA), developed by the Object Management Group (OMG), and is the broadest distributed object middleware available in terms of scope. CORBA runs on many platforms and also offers heterogeneity across programming language. It is considered by most experts to be the most advanced kind of middleware available and the most faithful to classical object-oriented programming principles.

```
<?xml version="1.0" encoding="UTF-8"?>
<methodCall>
  <methodName>flickr.echo</methodName>
  <params>
    <param>
      <value>
        <struct>
          <member>
            <name>name</name>
            <value><string>value</string></value>
          </member>
          <member>
            <name>name2</name>
            <value><string>value2</string></value>
          </member>
        </struct>
      </value>
    </param>
  </params>
</methodCall>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<methodResponse>
  <params>
    <param>
      <value>
        <string>
          <method>flickr.test.echo</method>
          <format>xmlrpc</format>
          <foo>bar</foo>
          ...
        </string>
      </value>
    </param>
  </params>
</methodResponse>
```

Figure 2.5: XML-RPC request (a) and response (b).

```
<?xml version="1.0" encoding="UTF-8"?>
<s:Envelope
  xmlns:s="http://www.w3.org/2001/06/soap-envelope"
  xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/1999/XMLSchema">
  <s:Body>
    <x:FlickrRequest xmlns:x="urn:flickr">
      <method>flickr.echo</method>
      <name>value</name>
    </x:FlickrRequest>
  </s:Body>
</s:Envelope>
```

```
<?xml version="1.0" encoding="UTF-8" ?>
<s:Envelope
  xmlns:s="http://www.w3.org/2001/06/soap-envelope"
  xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/1999/XMLSchema">
  <s:Body>
    <x:FlickrResponse xmlns:x="urn:flickr">
      <method>flickr.test.echo</method>
      <format>soap</format>
      <foo>bar</foo>
      ...
    </x:FlickrResponse>
  </s:Body>
</s:Envelope>
```

Figure 2.6: SOAP request (a) and response (b).

```
http://www.flickr.com/services/rest/?method=flickr.echo&name=value
```

```
<?xml version="1.0" encoding="UTF-8"?>
<rsp stat="ok">
  <method>flickr.test.echo</method>
  <format>rest</format>
  <foo>bar</foo>
  ...
</rsp>
```

Figure 2.7: REST request (a) and response (b).

2.6.2.2 DCOM/COM+

DCOM is a distributed object technology from Microsoft that evolved from its Object Linking and Embedding (OLE) and Component Object Model (COM). DCOM's distributed object abstraction is augmented by other Microsoft technologies, including Microsoft Transaction Server and Active Directory. DCOM provides heterogeneity across programming language but not across operating system as it is focused on the Windows platform. COM+ is the next-generation DCOM that greatly simplifies the programming of DCOM.

2.6.2.3 Java RMI

Java has a facility called Remote Method Invocation (RMI) that is similar to the distributed object abstraction of CORBA and DCOM. Since RMI is Java-specific, it can only be used on platforms running a Java Virtual Machine (that supports RMI).

2.7 Discovery Protocols

In an environment where mobile devices can enter and leave, there may be a need to *discover* devices and services they offer. Devices may notify their existence by means of a discovery protocol:

- As soon as a device enters the environment (e.g. the device is switched on), it can multicast a "Hello, I am a PDA with multimedia capabilities and..." message, which is received by one or more devices in the environment.
- A device can listen for incoming discovery requests and answer them with a similar message. In this case, a specific discovery request is sent by a controller device (control point).

A good example of a Service Discovery Protocol (SDP) is comprised in the Bluetooth protocol. If Bluetooth is switched on in a supporting mobile phone, the phone polls on regular intervals for incoming connections. A Bluetooth headset that is activated (and paired with the phone) notifies its presence to the phone and a connection is established automatically. If Bluetooth is enabled on the phone after the headset is turned on, the phone is still able to discover the headset by sending a discovery request.

Similar functionality is offered by higher-level protocols that are independent of the underlying (IP-based) network technology. Among these are Jini, Universal Plug and Play (UPnP), Salutation and Service Location Protocol (SLP). These protocols offer broader functionality than pure device/service discovery such as e.g. basic eventing. They are reviewed and compared in [17].

2.8 Summary

In this chapter we examined wireless communication protocols and stated that ethernet-based protocols are better suited than Bluetooth to interconnect the application logic and its remote interfaces. We differentiated between two network models: client-server and peer-to-peer. The former model is easier to achieve, since clients pose fewer requirements than peers which may be important for mobile platforms with limited networking support. Next, we discussed the HTTP protocol and introduced the promising XMLHttpRequest object used in web browsers. Then we focused on middleware solutions, and in particular on remote procedure calls and distributed objects. Especially the REST approach attracts the attention because of its simplicity. Also CORBA middleware may be worthwhile to integrate in (open source) UIDL renderers or wrappers, despite its complexity. Finally we got to device and service discovery, emerging techniques in the context of mobile and wireless computing.

Part II

Proposed Framework for Dynamic Distributed User Interfaces.

Chapter 3

Framework Preliminaries

3.1 Introduction

Developing a framework to support migratable user interfaces for mobile devices is a real challenge. Besides the already challenging features 'portability' and 'adaptability', a third feature is requested: 'migratability'. In the first place, the framework should be able to generate interfaces that can be rendered on and adapted to a wide range of devices. Next, the distribution of interface parts to mobile devices is of particular interest, since it allows users to interact with an application from anywhere a wireless connection is available. Moreover, there is an opportunity to exploit the potential of interconnected personal (mobile) devices to create a *distributed interaction space*. A distributed interaction space uses various resources that are available in the user's environment that can be accessed by the user. We distinguish two types of distributed interaction spaces: a *personal* interaction space where one person interacts with the application and a *collaborative* interaction space where different persons can use the (duplicated) distributed user interface parts to interact with the application. The latter requires a more complex supporting system since distributed locking of application data is necessary in this situation to ensure a consistent state during the run time use of the application.

We present a software system to support both types of interaction spaces. Our framework allows to extend native (existing) applications to support migratable UIs with minimal effort. An application offers certain *services* to its users that allow to execute certain tasks or visualize data. An example of the former are previous and next buttons (e.g. "view" service) to cycle through pictures in an image viewer and the latter could involve a thumbnail image of the pictures (e.g. "thumbnail" service). User interfaces that involve one or more of these services can be generated, distributed to and rendered on different devices (clients) in the interaction space. The distribution process (i.e. what service(s) should be distributed to what client(s)) can be controlled manually by the user or automatically by the system.

3.2 Preliminary Study and Related Work

One approach to achieve the migration feature is to extend an existing or create a new GUI toolkit with dynamic and migratable widgets. In [13], Grolaux et. al show a system for migratable user interfaces that relies on a particular software environment (Oz/Mozart) to support distribution. The application itself serves as a kind of server by combining receiver widgets that can receive events from migrated user interface parts. A specialized communication manager is responsible for redirecting event callbacks in this system. Since this and similar systems are toolkit-/platform-dependent, we prefer other solutions that are independent of the software environment of the system.

The use of description languages for UIs is one step towards a cross-platform solution. Approaches such as [2, 20, 5] and [19] have shown that a combination of XML-based UIDLs (section 1.4.1) and model-based interface design (section 1.5) are best suited to adapt the user interface for new contexts of use. In [2], Bandelloni and Paternò have shown that a web interface can be partially or completely migrated. Here, partial migration implies the web interface is split in two or more parts that each run on a separate device. This is accomplished by exploiting information that is available about the interactive system and by using a flexible language to describe the interface presentation. The former relies on the models that describe an interactive system [20] and the latter on a specific user interface description language [5].

Another option may be to integrate migration properties directly in the UIDL. Consider for instance UIML. We could introduce the tags `<splittable></splittable>` to indicate that widgets defined between these tags can be split from the rest of the UI and thus distributed to other devices (as suggested in [18]). However, this would break the current UIML specification. It would also imply that UIML renderers should understand these tags and handle migration or leave it to a server. Anyway, it is a UIDL specific way of defining the migration. We rather aim for a more general solution to capture the migration feature that is independent of the used description language. Thereby we assume that the UI is 'described' in XML format, e.g. in an XML-based UIDL or in XHTML[1].

It seems like a lot of migration supporting systems try to be all-in-one systems. They combine adaptation and migration of a UI by delivering a concrete, adapted UI to clients, generated from a (self-defined) specification language. We rather aim to split this functionality and allow the system to return an abstract UI specification (i.e. UIML) that can be passed to a context-aware rendering tool. As in ICrafter [24, 25], we focus on the distribution of the interface (concrete or abstract). The rendering is left to the client devices, i.e. a browser, UIML renderer, etc. The ICrafter system also supports the idea of generating and distributing UIs for services that are available in the environment. An Interface Manager (IM) is responsible for the generation of a UI for a certain service, requested by the user. A UI language generator for the service is downloaded from a repository and executed at the IM to produce e.g. Java Swing or HTML code.

[1] Because XHTML is widely supported by rather advanced renderers (browsers) and because of its well-known syntax, we will use it as running example format in the remainder of this text.

In our framework, we elaborate on the idea of schema-driven interface generation presented in [11]. Here, Kent Fitch introduces the use of a schema language for dynamically generating HTML forms. However, a self-defined (example) schema language is used. We believe it is better to stick to a current schema language specification that is already accepted by a lot of people and/or standardized by a well-known organization. [11] also illustrates the use of DOM, JavaScript and the XMLHttpRequest object; techniques that we will be using in our framework as proof of concept.

Chapter 4

User Interface Generation

4.1 Introduction

In this chapter, we outline an approach to generate concrete or abstract UIs that can be rendered on various devices. However, an interface cannot be generated out of thin air. There will always be some (abstract) UI specification to rely on. Two important considerations we make concerning this specification are:

1. The framework should be applicable to different UIDLs and not stick to a specific language.
2. It should be easy for the developer to design UIs that can be integrated with the system, without having to learn some exotic syntax.

What these considerations actually say is that a UI designer should be able to define a UI in UIML, XIML, XHTML, ... or any other (XML-based) UIDL and connect the UI with minimal effort to the framework. The term 'generating' a UI may sound odd in this context (as the specification to generate a UI from already defines UI code), but it will get clear in the next sections. After all, (parts of) the UI should be distributable.

4.2 Different Viewpoints on Migratable User Interface Development

We mention two strategies to develop a migratable UI; the order in which the actions take place is crucial here.

- 1. Design the interface
  2. Define migratable parts
- 1. Design migratable parts

2. Define the interface

Although the difference is quite subtle, both strategies assume a different reasoning about the UI. The former approach is probably the most intuitive, but after the first step we get a complete interface which we should split up again (virtually) in the second step. Notice that different defined parts are already related to each other after the first step (by design) and that the developer's task in the second step consists of indicating somehow that a collection of widgets form a migratable part.

If we apply the latter strategy, we end up with a number of separate non-related migratable parts after the first step. These parts can include for example a "main" part, a "playlist" part and a "settings" part for a multimedia player as shown in figure 4.1(a), 4.1(b) and 4.1(c) respectively. Figure 4.2 shows a visual representation of these parts, i.e. what they *could* look like. The second step, namely 'defining the interface', now consists of imposing constraints or in other words some *structure* on the parts. Since we use XML-based (part) descriptions, we can apply general techniques to impose a structure on XML documents. The common way to constrain the structure of XML documents is by means of an XML schema language.

```
<div id="main">
  ...
</div>
```
(a)

```
<div id="playlist">
  ...
</div>
```
(b)

```
<div id="settings">
  ...
</div>
```
(c)

Figure 4.1: Three interface parts for a multimedia player, defined in XHTML between `<div></div>` tags.

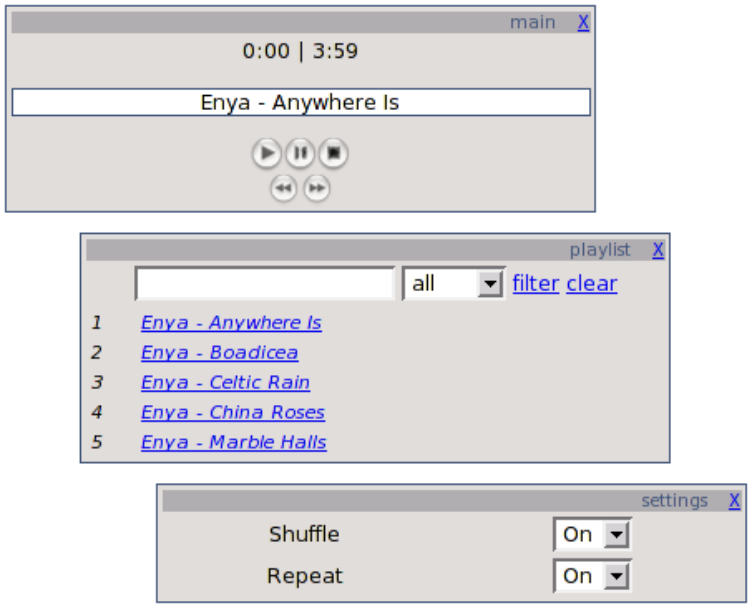


Figure 4.2: Description of the interface parts in figure 4.1, rendered in a browser.

We will use the second method to develop distributable UIs.

4.3 Schema-driven User Interface Generation

An XML document can be validated against a schema to check if it matches certain rules. The validation of XML documents incorporates different aspects:

- analyzing the structure of an XML document
- analyzing the content of each text and attribute node, independent of each other (datatype checking)
- analyzing constraints on relations between different nodes
- ...

Popular schema languages like Document Type Definitions (DTDs) and W3C XML Schema [40] are used in the first place to constrain and validate the *structure* of XML documents. Therefore a schema must contain at least information about the structure of instances it validates. If we look at a (not too complex) schema, we can easily think of an instance that will be validated by that schema. Moreover it is possible to design an algorithm that generates valid instances from a schema. The algorithm should understand the semantics of the concerning schema language and be able to convert them to XML instance data. Developing such an algorithm, however, is not a straigt-forward task because of various reasons. The most important reason is the fact that schemas can validate multiple and even an infinite number of XML documents. This is because traditional schema languages allow *choices*. Figure 4.3(a) shows a DTD incorporating a choice that validates an infinite number of instances and figure 4.3(b) lists an equivalent schema in the W3C XML Schema language. Valid instances include e.g. `<a><b>hello</b></a>`, `<a><b>hello</b><c>world</c></a>`, `<a><b>hello</b><b>world</b></a>`, ...

These examples show that the schema to instance algorithm should somehow decice what choice to make whilst constructing a valid instance. It can non-deterministacally generate a single instance based on random choices, but we want a deterministic algorithm to guarantee what the instance will look like. The algorithm should always return the same instance when it is run multiple times on a certain schema (with the same parameters). On the other hand, it is inefficient and likely impossible to generate all valid instances deterministically. And even if it were possible, what instance should the algorithm return?

4.3.1 RELAX NG

One of the advantages of RELAX NG [7, 29] over other schema languages is its simplicity and flexibility. RELAX NG focuses in the first place on validating the structure of XML

```
<!ELEMENT a (b*|c*)>
<!ELEMENT b (#PCDATA)>
<!ELEMENT c (#PCDATA)>
```

(a)

```
<schema>
  <element name="a" type="aty"/>

  <complexType name="aty">
    <sequence>
      <choice>
        <element name="b" type="string"
        minOccurs="0" maxOccurs="unbounded"/>
        <element name="c" type="string"
        minOccurs="0" maxOccurs="unbounded"/>
      </choice>
    </sequence>
  </complexType>
</schema>
```

(b)

Figure 4.3: DTD and corresponding W3C XML Schema document validating an infinite number of instances.

documents. Hereby, it differentiates itself from W3C XML Schema which offers different types of validation, resulting in a rather extensive and complex syntax. The syntax of RELAX NG is kept very simple what makes it easy to learn, read and write, but also to interpret by computers. Although its simple syntax, RELAX NG is powerful enough to describe practically every XML vocabulary based on wellformed XML 1.0 and namespaces.

RELAX NG comes in two versions that differ in notation: the RELAX NG XML syntax[1] and the RELAX NG simple syntax[2]. Tools are available to convert both syntaxes to each other. We will only consider the XML variant as it is the easiest to parse.

Since RELAX NG also allows to define constraints for text values, it is possible to define a schema for an XML file that only validates that specific file. At least, if the XML file does *not* contain mixed content. One of the drawbacks of the current RELAX NG specification is that it does not allow to constrain text values in mixed content nodes. Consider for instance the XML document in figure 4.4; the `p` node contains both text ("hello") and an element (`a`) what is called "mixed content". The text value "hello" cannot be constrained here. Because of this, we will assume that user interface related XML documents may *not* contain mixed content. Hereby we put a constraint on the XML 1.0 specification [34], but one that is acceptable. In many cases, mixed content can be avoided and often it even is an ecouraged design principle to avoid it.

Thus, we can define a unique RELAX NG schema for an XML file by constraining each element, attribute and text value in the XML file given that it does not contain mixed content. This means an interface part for a service can be defined in XHTML, e.g. between

[1] `http://www.relaxng.org/tutorial-20011203.html`
[2] `http://www.relaxng.org/compact-tutorial-20030326.html`

```
<p>
  hello
  <a href="...">world</a>
</p>
```

Figure 4.4: XML node with mixed content.

`<div></div>` tags and converted to RELAX NG schema code. The XHTML to RELAX NG schema conversion (algorithm) is very simple as figure 4.5 (partially) illustrates. Furthermore, the original XML instance can easily be regenerated from the schema.

```
<define name="main">
  <element name="div">
    <attribute name="id">
      <value>main</value>
    </attribute>
  </element>
  ...
</define>
```

Figure 4.5: RELAX NG definition of the interface part in figure 4.1(a).

We can create RELAX NG schema code for each service an application offers and give that service a name in the schema[3]. This allows the inclusion of service definitions of an application in a RELAX NG schema. This schema describes, for example, a web interface and constrains the `html`, `head`, `body`, `div`, ... elements. It can also constrain the included service definitions by referring to them inside RELAX NG patterns. Figure 4.6(a) illustrates how we can define that service S_1 and S_2 are both optional, while figure 4.6(b) shows that service S_1 and S_2 are optional but should always appear together. This can also be interpreted as "S_1 and S_2 *may* split and *may* appear in an interface instance" and "S_1 and S_2 *may* only appear together in the interface instance" respectively. The RELAX NG construction in figure 4.6(c) says that either service S_1 or service S_2 should be included in the final instance. We can interpret this as "S_1 or S_2 *must* appear in the interface instance". Notice that the introduction of optional and choice related patterns introduces different *paths* that can be followed while generating an actual instance from the schema. Each path results in a specific user interface that incorporates a number of services.

Figure 4.7 lists a RELAX NG schema that includes (lines 34 – 39) the service interface code from figure 4.1 wrapped in named RELAX NG patterns, like in figure 4.5. Lines 21 – 31 relate and constrain the appearance of the services. What this schema actually says is that the "main" service *may* appear and that or the "playlist" or the "settings" service *must* appear in a valid user interface instance. Also, if the "main" service is selected, it will appear

[3]RELAX NG supports so called "named patterns". A part of a RELAX NG schema can be given a name by which it can be referred in the rest of the schema.

```
<optional>
  <ref name="S1"/>
</optional>
<optional>
  <ref name="S2"/>
</optional>
```
(a)

```
<optional>
  <ref name="S1"/>
  <ref name="S2"/>
</optional>
```
(b)

```
<choice>
  <ref name="S1"/>
  <ref name="S2"/>
</choice>
```
(c)

Figure 4.6: Three different RELAX NG constructions to constrain the appearance of services.

above the "playlist" or "settings" service and there will be a blank line (`<br/>`) between both interface parts. It should be clear that different constraints can easily be put on the referred services by 'playing' with the appropriate RELAX NG patterns. It can e.g. be defined that the "main" and "playlist" service part may appear next to each other by placing them in an HTML table.

Note that this schema is quite restrictive, since the "playlist" or "settings" part *must* appear in any generated UI. In general, migratable UIs are preferred to be more flexible, i.e. users may wish to hide the playlist which is not allowed according to the schema. However, this schema is well suited as example in the next section, where we outline our algorithm for generating XML (user interfaces) instances from a RELAX NG schema.

4.3.2 RELAX NG-based Schema to Instance Algorithm

There is a publicly available tool called Sun XML Instance Generator[4] (written in Java) that generates instances from different schema languages, also RELAX NG. However, it generates random instances and it is intended for testing purposes. Moreover, it relies on old and deprecated JDK classes. We did not find any other relevant (open source) tools or algorithms to build upon. Therefore we wrote our own algorithm from scratch.

The algorithm operates in two phases. After the first phase, user interaction is requested before going on with the second phase.

Phase 1

Input $I = S$ with S the schema to generate (an) instance(s) from.

First, the schema S is parsed and read into a DOM object [44] (in main memory). The algorithm traverses the XML tree in depth-first order and constructs a new (customized) tree from it. It does so by calling a specific *evaluator* for each RELAX NG pattern node it meets. An `element` node triggers an `ElementEvaluator`, an `optional` node an `OptionalEvaluator` and so on. Each evaluator calls on its turn evaluators for nested pattern nodes and returns an

[4] `http://www.sun.com/software/xml/developers/instancegenerator/`

`EvalNode` object (table 4.1). An `EvalNode` object stores extra information in an `EvalRecord`, compared with a traditional DOM node (table 4.2). This extra information mainly consists of a list of possible paths that can be followed from the context node (table 4.3).

`EvalNode`	
text	Text value copied from the DOM tree, like the name of an element, attribute, ... as it would appear in an instance.
record	`EvalRecord`
parent	Pointer to the parent node.
child nodes	Pointers to child nodes.

Table 4.1: `EvalNode` object.

`EvalRecord`	
paths	List of `EvalPath` objects.

Table 4.2: `EvalRecord` object.

`EvalPath`	
services	List of services that *will* be reached from the context node, when following this path.
choice indexes	List of indexes that indicate what choice path (child node) to choose when arriving at a choice node.
optional flags	List of boolean values that indicate whether an optional node should be chosen or not.

Table 4.3: `EvalPath` object.

The construction of this tree is simulated in figure 4.8[5] for the schema in figure 4.7. The three `group` nodes in the third level match the service references on lines 23, 29 and 30 in the example schema. Notice the `ns` attribute in the `ref` nodes. The namespace attribute could for instance hold the name of the application. This attribute is a hint for the algorithm to indicate that we refer to a specific service user interface (part). If it is left out, the algorithm cannot know that we are referring to a service, since it could also be a schema construction. For example, `<ref name="main"/>` can refer to a defined element named "main" that has nothing to do with a service.

The references can be replaced by the definition of the RELAX NG code they refer to (figure 4.5 and similar). Within the `<definition></defintion>` tags, one or more child nodes may occur; in our example this is a single element node (`div`). The child nodes together are evaluated as a group by a `GroupEvaluator` that returns the `GroupNode` objects (nodes) shown in the figure. The child nodes generated in a lower level (combined with node

[5]We left out the parent and child node pointers.

data from the DOM tree) are used as input to generate the parent node in a higher level. The depth-first traversal recursively backtracks to the higher level and when each of a node's child nodes are evaluated, the node itself is constructed. Hereby we differentiate between the following RELAX NG container patterns:

- `optional`: The child nodes of an optional node *may* but *must not* occur. This introduces two possible paths: one that does not choose the child nodes to be included, and one that does so. The paths are marked with an optional flag "false" and "true" respectively (second level in figure 4.8 on the left).
- `choice`: One of the child nodes of a `choice` node *must* be chosen. Thus, there are n possible paths where n is the number of child nodes. Each path is marked with the according child index i where $i = 0, \ldots, n-1$ (second level in figure 4.8 on the right).
- `group`: A `group` indicates that each child node must occur in the order specified in the XML file. This implies that record data from the child nodes should be merged. In particular, all possible path combinations should be calculated. Notice that the `<group></group>` tags may be specified explicitly, but often it will not because RELAX NG implicitly assumes that child nodes form a group, unless specified otherwise (e.g. by the `interleave` pattern). Thereby child nodes of e.g. an `element` node can be evaluated as a group (first level in figure 4.8).

Other RELAX NG container patterns namely `interleave`, `mixed`, `zeroOrMore` and `oneOrMore` are mapped on the above mentioned patterns, since we believe they do not add extra (useful) power for the generation of instances. The mappings are described in table 4.4. Notice that we do *not* break the RELAX NG specification by these mappings! We only stick to a subset of the RELAX NG expressive power to put constraints on XML data, i.e. user interface parts.

In figure 4.8 we only listed the most important part of the tree for the schema in figure 4.7 (to keep the overview). The missing nodes on the bottom of the tree only contain values like the name of elements (`div`, `p`, `a`, ...), attributes (`id`, `class`, ...) or text values. Their `EvalRecord` fields are expected to be empty. It has no sense to use choice related patterns in a strict RELAX NG definition of XML code of an interface part (figure 4.5). The `html` root node left out on the top of the tree will have the same `EvalRecord` as the RELAX NG encoded `body` node in the first level. Only its text value will differ.

The root node now contains information about all valid paths that result in a specific instance matching the schema. Moreover a list of possible service combinations can easily be retrieved from the root node. From our example schema a UI for the "playlist" and "settings" service separately as well as for the "main" and "playlist" or "main" and "settings" service together can be generated. Other combinations (e.g. only the "main" service) would violate the constraints specified in the schema.

EvalPath	
`interleave` → `group`	The `interleave` pattern indicates that the order of the nodes nested between it does not matter[6]. We choose the default order by substituting `interleave` by `group`.
`mixed` → `group`	The `mixed` pattern is nothing more than a shorter notation for the interleave of text and element nodes (mixed content). Therefore we can also replace the `mixed` pattern by the `group` pattern for the same reason as with `interleave`.
`zeroOrMore` → `optional`	We try to fulfill the semantics of this pattern in the least possible way. This means that the content may or may not appear which resembles the semantics of the `optional` pattern.
`oneOrMore` → `group`	The same reasoning as with the `zeroOrMore` pattern applies. The `oneOrMore` pattern can be restricted to the `group` pattern by avoiding repetitions of the node's content.

Table 4.4: RELAX NG pattern mappings.

To go on with phase 2, one of the possible paths should be chosen. A path can either be selected manually by the user (section 5.3) or automatically by the system (section 5.4).

Phase 2

Input $I = (T, p)$ with T the tree constructed in phase 1 and p the selected path.

To generate an instance, it suffices to traverse T by following the path indications stored in p and transform data values in each node to XML code. For simplicity reasons, the algorithm operates in two passes in this phase. In the first pass, the tree is *pruned*, i.e. `optional` and `choice` nodes are pruned. While traversing the tree in depth-first order, each `optional` node that has a corresponding "false" flag in p is removed. If a schema tree contains three `optional` nodes, then each `EvalPath` object in the root node will have three optional flags corresponding with these nodes. The same applies for `choice` nodes, but instead of removing such a node it is replaced by one of its child nodes with index i where i is also deduced from p. As soon as all choice related nodes are eliminated, the tree represents a fixed instance. In the second pass, the algorithm traverses T for the second and last time to generate the actual XML code.

Evaluation

Each phase of the algorithm presented here can be evaluated separately. The first phase should only be executed once. The tree outputted by this phase can be stored in main memory and duplicated and pruned concurrently in the second phase. Unfortunately the number of possible paths can grow fast, since all possible options are combined. Thus, the more (possibly nested) choice related patterns are used in a schema, the more paths will be created. However, this is no big deal because an application's interface is expected to offer only a limited number of migratable parts (services). A limited number of services also means a limited number of schema defined constraints and thus possible paths. It is very difficult to calculate an average cost for this pass as it depends in the first place on the structure of the input schema and less on its length.

The cost of the second phase is dictated by the cost of traversing a tree with n nodes in depth-first order. This only requires linear time in the number of nodes n. The second phase could also be executed in a single pass by traversing the tree only once. However, we obtained for two passes because of practical reasons (e.g. easier to debug). The performance gain of a single pass algorithm would be negligible, especially compared with the time it takes to render the generated interface (a few miliseconds versus a few tenths of a second or more, depending on the renderer).

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <grammar xmlns="http://relaxng.org/ns/structure/1.0" ns="http://www.w3.org/1999/xhtml
       ">
4      <start>
5          <ref name="html"/>
6      </start>
7      <!-- definition of "html" node -->
8      <define name="html">
9          <element name="html">
10             <ref name="head"/>
11             <ref name="body"/>
12         </element>
13     </define>
14     <!-- definition of "head" node -->
15     <define name="head">
16         ...
17     </define>
18     <!-- definition of "body" node -->
19     <define name="body">
20         <element name="body">
21             <!-- UI parts -->
22             <optional>
23                 <ref name="main" ns="xmms"/>
24                 <element name="br">
25                     <empty/>
26                 </element>
27             </optional>
28             <choice>
29                 <ref name="playlist" ns="xmms"/>
30                 <ref name="settings" ns="xmms"/>
31             </choice>
32         </element>
33     </define>
34     <!-- definition of "main" part -->
35     <include href="main.rng"/>
36     <!-- definition of "playlist" part -->
37     <include href="playlist.rng"/>
38     <!-- definition of "settings" part -->
39     <include href="settings.rng"/>
40 </grammar>
```

Figure 4.7: RELAX NG schema defining a web interface for a multimedia player.

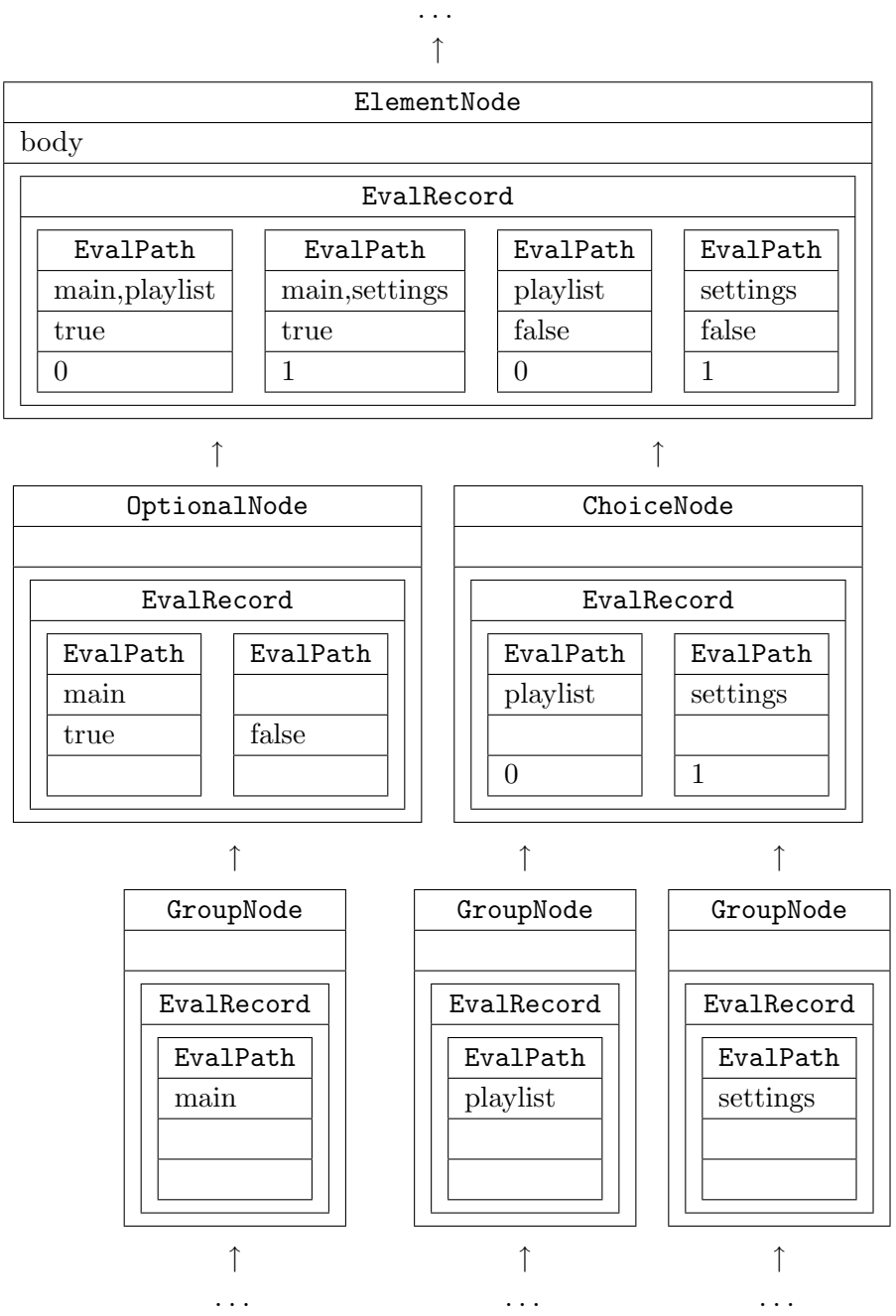


Figure 4.8: Bottom-up construction of a tree representing the information in a RELAX NG schema.

Chapter 5

User Interface Distribution

5.1 Introduction

An algorithm for generating user interfaces is one thing, but it still should be invoked with the right parameters (e.g. schema, path, ...). Furthermore the generated UI should be 'transferred' to the client device whilst both parties should be able to communicate with each other. For this purpose, we built a light-weight HTTP-based daemon into our framework. The daemon supports traditional methods like GET, POST, HEAD, DELETE and PUT to exchange data and messages and is discussed in detail in chapter 6. Hereby we combine schema-driven UI generation (section 4.3) with the REST approach (section 2.6.1.3). We shall refer to the combination of both core modules as the *Interface Distribution Daemon* (IDD). This chapter elaborates on the IDD and explains how the distribution of an application's UI is realized.

5.2 Distributed Interaction Spaces

Before a distributed session can take place, one or more applications should register with the IDD and transfer the necessary data such as the RELAX NG schemas to generate UIs from. An application should indicate what services it offers and make them available to clients. A service can be marked as "distributable to only one client" or "distributable to multiple clients". The former is applicable for most control-driven UIs, to avoid that clients can execute conflicting actions. Once a service is taken, it is marked as "unavailable" in this setting. The latter setup allows a service to be duplicated among different devices. This can be useful to show a thumbnail image on each device in the interaction space or general information like the current song playing.

The registration process is handled over the HTTP-protocol, by sending the appropriate messages to the IDD. Clients can register in a similar way, but rather than sending raw messages we opted for a user-friendly web interface that simplifies the registration proces. The

web interface can be rendered and used in any web browser supporting (basic) JavaScript. It takes the user through three to four stages. Figure 5.1 shows a screenshot of each stage.

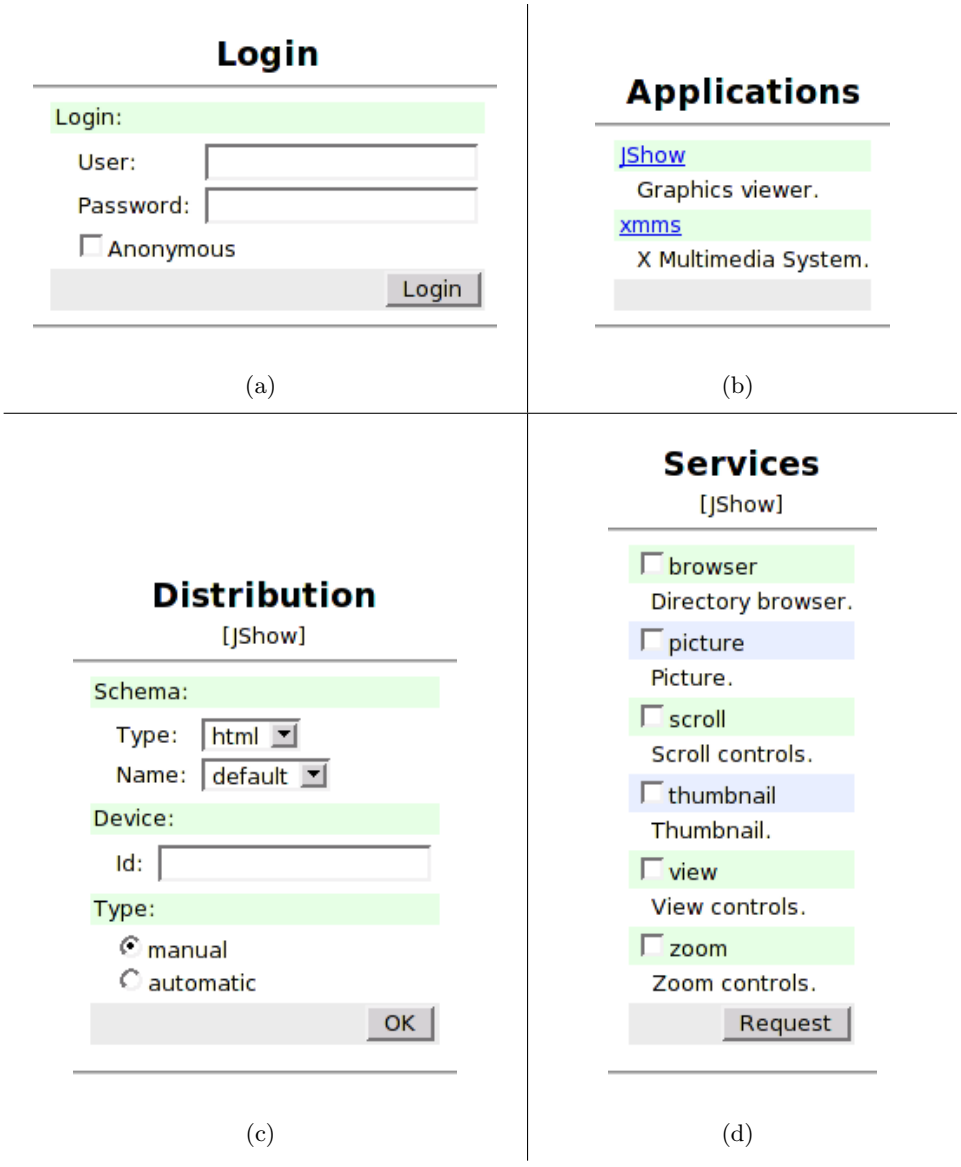


Figure 5.1: Web interface for registering clients and requesting service user interfaces.

First, the user points the device's web browser to the IP-address of the IDD or its URI when a Dynamic Name Service (DNS) is in use in the environment. The IDD returns the login dialog (figure 5.1(a)) and the user can login with username and password. The user now gets an overview of the applications that have registered with the IDD (figure 5.1(b)). She/he can select one and possibly choose between different schema "types" in the next stage (figure 5.1(c)). Since a schema can constrain *any* XML-based vocabulary, an application can provide schemas for different languages, e.g. one that generates XHTML code, one for UIML, ... Apart from the schema type, the user can also select the schema's "name". This is because an application may provide different schemas for a single UI language, say XHTML.

This allows skinning, but it can also offer the option to generate a tiny or a more advanced concrete UI for certain services. Next, the user can choose between manual or automatic distribution[1]. We distinguish three different distribution types: user-driven, system-driven and continuous distribution that will be discussed in detail in the remainder of this section. The main difference between the former two is in the applicant of the services. The user may choose to request services manually by selecting them in the service dialog (figure 5.1(d)[2]). Notice that a UI schema for an application may offer only a subset of the application's services. This will typically be the case with 'special-purpose' skins, e.g. for devices with special requirements. The user can also let the system decide what services to include and generate/distribute a UI automatically. Hereby the IDD may be guided by different profiles that include information about the schema, the user and the device.

Although the web interface is a convenient instrument for registering clients manually, it can be difficult to register a number of clients by hand. Suppose for example that a user wants to setup an interaction space with all client devices in the room involved. It is more convenient for the user if she/he can tell to the system "look for available devices and register them automatically", especially if some devices are used to visualize data rather than to control an application. This can be realized by multicasting a discovery request that clients may answer by sending a registration message to the IDD (see section 2.7). Once clients are registered, the IDD can calculate an optimal distribution (possibly guided by the user) and migrate useful UIs to the appropriate devices.

5.3 User-driven Distribution

User-driven distribution relies on the initiative of the user: the end-user connects with the IDD and requests an interface for her/his personal interaction space by choosing one or more services, offered by a registered application. The IDD looks for a path to generate a UI from (see section 4.3) that contains all the selected services. Hereby the IDD may operate in *strict* mode, in which a path should contain exactly the selected services and no others. On contrary, *non-strict* mode allows a path to have services other than the selected ones. By default, strict mode is applied. If no path is found, the IDD informs the client of this, and possibly suggests some viable service combinations, based on the chosen services.

In many cases, user-driven distribution is preferred since the end-user has full control over the distribution of a UI to device in her/his personal interaction space.

[1] In the case of automatic distribution, the user may also want to specify the device she/he is using (see section 5.4.1)

[2] Services marked in green are available, in red are unavailable (taken by another client) and in blue can migrate to multiple clients at once.

5.4 System-driven Distribution

With system-driven distribution the IDD itself decides what services are distributed to what clients. The IDD will typically broadcast a discovery request that may trigger clients to register them on the IDD and choose for *automatic distribution*. However, users can also opt for automatic distribution through the IDD's web interface.

Once the IDD knows about all the clients that registered for a system-driven distribution session, it can trigger its distribution algorithm. The algorithm tries to perform an *optimal* distribution of services among the clients. Notice that 'optimal' is not formally specified. For one person, optimal distribution may entitle to distribute as many services as possible to all devices. Another person may prefer to distribute only those services that fit well to the user's PDA screen. Still, many other scenarios are conceivable. Therefore it is crucial that the IDD does not define itself what optimal distribution should involve, but leave that decision to the end-user(s). The combination of *profiles* and *patterns* allows us to define various key information to feed the algorithm with and guide the distribution process. We will now give a formal definition of the system-driven distribution problem in the context of our framework and then explain the role of profiles and patterns in a solution for it.

> Given a collection of paths[3] $V_P = \{P_1, P_2, ..., P_n\}$ and a collection of devices $V_D = \{D_1, D_2, ..., D_m\}$, try to assign to each device $D \in V_D$ a path $P \in V_P$ in a way that none of the assigned paths $P_i, P_j \in V_P$ with $i \neq j$ are conflicting. Two paths cause a conflict if they share a service that is only distributable to a single device.

5.4.1 Profiles

Profiles are used to check if a given service *can* migrate to a given device. We distinguish three different profile types that are closely related with each other:

- **Device profile**: A device profile specifies the characteristics of a device, like its screen size, whether is can render images, if it runs a Java VM and so on.
- **Service profile**: A service profile defines the requirements a device should fulfill in order to have the service distributed to it. It can e.g. define that a minimum screen resolution of 320x240 pixels is required.
- **User profile**: A user profile indicates which requirements in the service profile are relevant to the user and which are not. For example, a user profile may define that the screen size is irrelevant and thereby override the service profile.

With the help of these profiles, the following subproblem can be solved:

[3] We refer again to section 4.3 where we discussed paths in the context of a RELAX NG schema used to generate a UI. Table 4.3 gives an idea of the computer representation of such a path.

> Given a collection of services $V_S = \{S_1, S_2, ..., S_n\}$ and a collection of devices $V_P = \{D_1, D_2, ..., D_m\}$, output for each device $D \in V_P$ the subset of services ($\subset V_S$) that can be distributed to D according to the information in the various profiles.

We will now take a closer look at the different profiles.

5.4.1.1 Device profile

For the device profiles, we built upon CC/PP profiles [43] as a proof of concept. These profiles describe the capabilities of a device in Resource Description Framework (RDF) [36] syntax. RDF is written on top of XML (further abstraction) and is one of the core technologies used to realize the Semantic Web [3]. The RDF language is used to annotate data, give it well-defined meaning. Hereby the notion of ontologies is important, as an ontology represents a concrete domain that can be queried by web agents. For example, an ontology may represent the domain of computer devices and define that each device has a processor, but that there is a difference between a mobile phone, a PDA, ..., although they also have some things in common like a display with a certain resolution. Notice the importance of this *meta-information*: an agent could query a computer network using the ontology and ask e.g. for all devices that have a minimum screen resolution of 320x240 pixels! Nowadays, the semantic web and everything that is related with it is a hot topic. Since RDF-based profiles fit in the semantic web picture, they have great potential.

Figure 5.2 lists a (partial) CC/PP profile for a generic PDA. However, the example does not comply with the current CC/PP Recommendation. This is because we splitted

```
<prf:ScreenSize>320x240</prf:ScreenSize>
```

as indicated by the W3C Recommendation in

```
<prf:ScreenSizeWidth>320</prf:ScreenSizeWidth>
<prf:ScreenSizeHeight>240</prf:ScreenSizeHeight>
```

to become two tags specifying an integer value. The reason for this will be clear when we arrive at the service profiles.

5.4.1.2 Service Profile

A service profile actually *constrains* a device as it says that it should be able to render images and/or have a minimal screen size of 320x240 pixels. We want to be able to tell if device D is suited for service S. We can formulate this problem as follows: check if a device profile (D) matches a service profile (S). This again resembles validating an instance against a schema. The instance is the device profile and is already defined, while the schema consists of the

```
<?xml version="1.0" encoding="UTF-8"?>

<RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:prf="http://www.wapforum.org/UAPROF/ccppschema-20000405#">
  <rdf:Description ID="Profile">
    <prf:component>
      <rdf:Description ID="HardwarePlatform">
        <prf:ScreenSizeWidth>320</prf:ScreenSizeWidth>
        <prf:ScreenSizeHeight>240</prf:ScreenSizeHeight>
        <prf:BitsPerPixel>8</prf:BitsPerPixel>
        <prf:ColorCapable>Yes</prf:ColorCapable>
        <prf:TextInputCapable>Yes</prf:TextInputCapable>
        <prf:ImageCapable>Yes</prf:ImageCapable>
        <prf:Vendor>Generic PDA</prf:Vendor>
        ...
      </rdf:Description>
    </prf:component>
    ...
  </rdf:Description>
</RDF>
```

Figure 5.2: CC/PP profile for a PDA (device profile).

service profile. But, what schema language is preferred? Since device profiles are defined in the RDF format, a logical step is to look after schema languages for RDF. Among them are RDF Schema (RDFS) [35] and the more powerful Web Ontology Language (OWL) [38]. However, there are two objections against these languages:

1. They are still poorly supported and (offline) maintained validators are rare, not to say unavailable.

2. We aim to support all kind of device profiles, also non-RDF-based profiles.

Therefore we descend one layer on the abstraction ladder and arrive at the XML level. A schema language for XML can also constrain RDF instances because RDF conforms to the XML standard. So far, we have had good experiences with RELAX NG and again, it will proof its strength here. First, we designed a schema prototype that validates *all* well-formed CC/PP profiles, shown in figure 5.3. Though the schema in the figure is incomplete, it should be clear that it can be constructed. Note that the schema utilizes of a *datatype library*, namely the W3C XML Schema Type Library. Datatype libraries are some kind of extension to RELAX NG, to support datatype validation. Hereby it is for instance possible to check if a value is a valid integer. The W3C XML Schema datatypes that can be used in a RELAX NG schema are the predefined W3C XML Schema types, defined in the Recommendation. Furthermore, custom datatype libraries can be created and integrated with a supporting RELAX NG validator. Also, restrictions can be applied to these datatypes using the RELAX

NG `param` pattern, so customization is possible. Figure 5.4 list a service profile, i.e. a more restricted version of the RELAX NG schema in figure 5.3. The profile includes the default CC/PP definitions and overrides the definitions it wants to constrain more. For example, the definition of the `prf:ScreenSizeWidth` element has been redefined, in order to accept only those device profiles having a value of at least 320. See here the reason why we divided the CC/PP proposed `prf:ScreenSize` element in a width and height component: it is much easier to validate two separate integer values than a '320x240' construction. Nevertheless, a datatype can be created to validate the latter.

5.4.1.3 User Profile

The service and user profiles may seem to be contradictory at first sight. The first specifies a number of requirements, while the second specifies which requirements should be taken into account and which not. The reason for this is to avoid that the IDD does not distribute certain services to a client because the system discovers that the client's screen size is too small, whilst the user believes it is large enough. Assume for instance that service S, in general, really needs a screen resolution of 320x240. However, we bought a brand new smartphone with a screen size of only 160x120 pixels, *but* the phone supports new scaling techniques that allow it to render a UI with S included like a charm. Thereby the minimum screen size constraint in the service profile is a thorn in the flesh since it disallows the distribution of S to the phone. One option to overcome this, is to set a fake screen resolution in the device profile, but this may result in unwanted side effects. A better solution is to setup a user profile and specify that the screen size should not be taken into account (for a particular user and/or service).

5.4.2 Patterns

Thanks to the information in the profiles, the system-driven distribution algorithm is able to tell if a service S can migrate to a device D. Thus, the algorithm can calculate a *possible* distribution, but it still has no clues about an *optimal* distribution. This is where the distribution patterns come in. A pattern rule resembles a regular expression that guides the distribution process in a session with m devices. Suppose for example the following rule:

```
1:S1,S2,(S3|S4)
```

that applies to an interaction space with only one device (D) involved (indicated by the "1" before the ":"). The rule states that the IDD should *try* to distribute services S_1, S_2 and S_3 or S_4 to the device. Notice the stress on *try*: there are no guarantees that the pattern can be matched! After all, service S_1 may not be migratable to D at all, according to the device/service/user profiles. In that case, the IDD shall distribute the services S_2 and S_3 or S_2 and S_4 to D. Thus, the patterns are hints to guide the distribution process, but the profiles still have absolute priority.

A pattern rule for two or more devices is a little more complicated, e.g.

```
<?xml version="1.0" encoding="UTF-8"?>

<grammar xmlns="http://relaxng.org/ns/structure/1.0"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:prf="http://www.wapforum.org/UAPROF/ccppschema-20000405#"
  datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">
  <start>
    <element name="RDF">
      <ref name="Profile"/>
    </element>
  </start>
  <!-- Profile -->
  <define name="Profile">
    <element name="rdf:Description">
      <attribute name="ID">
        <value>Profile</value>
      </attribute>
      <interleave>
        <ref name="HardwarePlatform"/>
        <ref name="SoftwarePlatform"/>
        ...
      </interleave>
    </element>
  </define>
  <!-- HardwarePlatform -->
  <define name="HardwarePlatform">
    <element name="prf:component">
      <element name="rdf:Description">
        <attribute name="ID">
          <value>HardwarePlatform</value>
        </attribute>
        <interleave>
          <ref name="ScreenSizeWidth"/>
          <ref name="ScreenSizeHeight"/>
          <ref name="ColorCapable"/>
          ...
        </interleave>
      </element>
    </element>
  </define>
  ...
</grammar>
```

Figure 5.3: RELAX NG schema skeleton that validates any valid CC/PP profile.

```
<?xml version="1.0" encoding="UTF-8"?>

<grammar xmlns="http://relaxng.org/ns/structure/1.0"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:prf="http://www.wapforum.org/UAPROF/ccppschema-20000405#"
  datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">
  <include href="ccpp.rng">
    <!-- override ScreenSizeWidth -->
    <define name="ScreenSizeWidth">
      <element name="prf:ScreenSizeWidth">
        <data type="integer">
          <param name="minInclusive">240</param>
        </data>
      </element>
    </define>
    <!-- override ScreenSizeHeight -->
    <define name="ScreenSizeHeight">
      <element name="prf:ScreenSizeHeight">
        <data type="integer">
          <param name="minInclusive">240</param>
        </data>
      </element>
    </define>
  </include>
</grammar>
```

Figure 5.4: RELAX NG schema that further restricts the schema in figure 5.3 (service profile).

```
2:S1,S2;S3,S4
```

where the ";" separates the two patterns to aim for. This rule applies to a distribution space that involves two devices, say D_1 and D_2. The extra difficulty lays in the following choice: what device should try to match what pattern? Here, D_1 could strive for the `S1,S2` pattern, but what if none of these services can be distributed to D_1, but only to D_2? It then figures that it is better to let D_2 match the first pattern and let D_1 aim for `S3,S4` as more services will be distributed. Therefore we first calculate all possible permutations of m devices/patterns. There are $m!$ such permutations and the one with the (first) highest sum of distributed services will be chosen.

5.5 Continuous Distribution

When a user is engaged in a distributed interaction session, changes in her/his interaction space may trigger dynamic changes in the distribution of the user interface. Two main causes of environment changes can be distinguished in this context: client devices entering or leaving the interaction space.

When a client leaves the interaction space its registration with the IDD will be canceled. This implies that the services held by the client, come available for other clients. These services may be distributed automatically among the devices in the interaction space without disturbing the execution of the application. Therefore, the IDD looks for clients in the environment that satisfy the requirements of one or more services and migrates the updated service UI(s).

When a new client enters the environment it announces its presence and registers with the IDD. If the new client device better fulfills the requirements for a particular service already running in the interaction space, the IDD can decide to migrate the service interface to the new client. Hereby the IDD relies on profiles and patterns, like in system-driven distribution (section 5.4).

Important to notice is that continuous distribution in an interaction space implies some usability issues. This has to do with the fact that the user interface changes while the user is interacting with it. This may be very frustrating for the user when client devices enter and leave the environment all the time. To make this process more manageable, the user may define some constraints for this process in her/his user profile. This, however, is outside the scope of our work and will be a subject for further research.

Chapter 6

Architecture

6.1 Introduction

In the previous chapters, parts of the architecture were already introduced. This chapter glues the various parts together and gives a global overview on the architecture of the framework. The IDD is the central entity (server) and both applications as client devices (clients) are connected with the IDD. This is shown in figure 6.1. The programming language of the applications nor the operating system on which they run does matter. Client devices can be any computing device: PDA, desktop PC, tablet PC, laptop with a projector attached, etc. With this picture in mind, we are ready to take a closer look at the architecture of the IDD and explore how distributed user interfaces can talk with the application logic.

6.2 RESTful communication

The architecture is based on the REST principle, outlined in section 2.6.1.3. Clients and applications can communicate with the IDD by exchanging HTTP messages. The built-in HTTP server processes an HTTP request and replies to it with an HTTP response message. The daemon is developed with the HTTP 1.1 specification [15] in mind and currently supports the GET, POST, HEAD, DELETE and POST methods. One of our design goals is to keep the framework as light-weight as possible which is also reflected in the HTTP daemon. The implementation only relies on the default Java classes as provided with the JDK 1.5, and does not support PHP, CGI, Java Servlets, ... However, we do need a way to process data sent to the server, e.g. parameters in a URI, form data, XML data in a message body, etc. Therefore we developed a module system that is similar to Apache's module system[1] and that does not require external software like Apache Tomcat[2] (to host Java Servlets).

[1] `http://modules.apache.org/`
[2] `http://jakarta.apache.org/tomcat/`

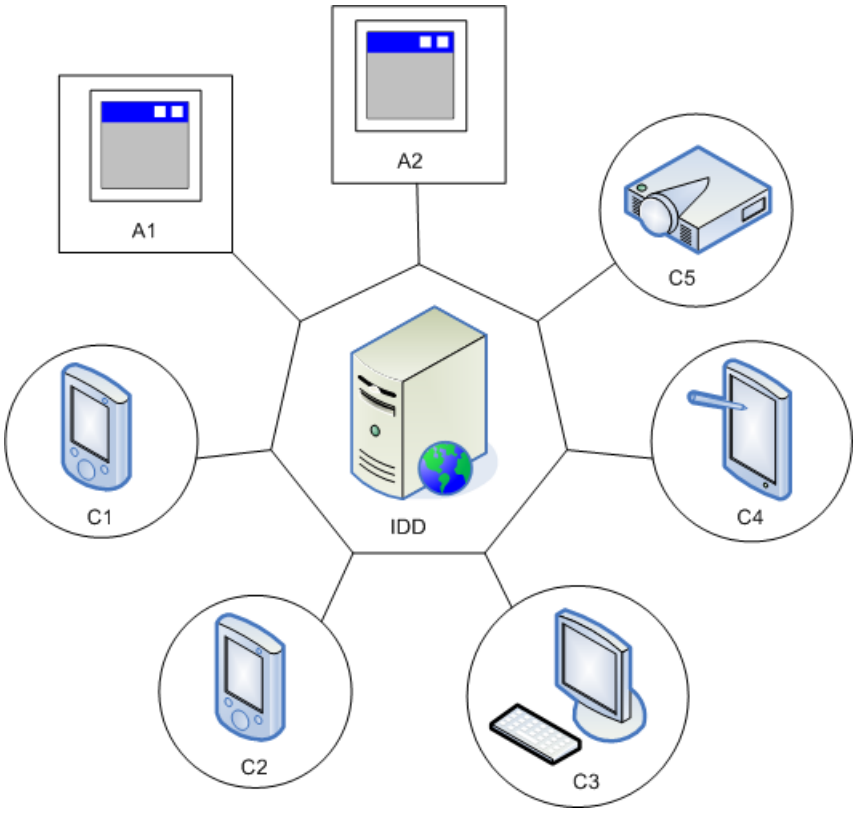


Figure 6.1: Architecture: applications, client devices, IDD.

Modules

A module can be triggered from within an HTTP request by the GET or POST method. Data is *posted* directly to the module which will be executed with the data of the request data as input. The following HTTP header shows how to trigger a module:

```
POST /_my_mod HTTP/1.1
```

The "_" before "mod_name" tells the IDD that it should look for a module named "my_mod", instead of a file in the webroot. The HTTP daemon reads and parses the request and passes its data to the module. In fact, modules are nothing more than regular Java classes that implement the `Module` interface. The modules are loaded dynamically by a `ModuleLoader` (a derived Java `ClassLoader`) when requested. Modules are advantageous because:

- They can be coupled tightly with the framework itself (both Java code). This is a must because the modules trigger all vital actions like converting a schema to an instance or registering clients/applications.
- They can react fast because they are precompiled, just like Java Servlets.

- They provide a means to extend the system. A module can be compiled separately against the `Module` interface and placed in the appropriate directory (in binary form) and used without having to recompile the framework or even restart it when it is in use.

In the next few sections, we give an overview of the currently available modules. A module replies a request with an OK response (status code 200) by default, unless specified otherwise. The client, application and session ids mentioned below are HEX-encoded, random generated hash values. The module overview can be seen as 'the REST API of the IDD'.

mod_register_client

Input:	`user`, `password`
Output:	`clientId`

The module registers a client on the IDD given that a valid username and password are provided. A unique client id is generated and returned in a cookie.

mod_unregister_client

Input:	`clientId`
Output:	–

The module unregisters the client with the given id.

mod_register_application

Input:	`appName`, `appKey`
Output:	**appId**

The module looks for an application map with the given application name in `/applications`. If found, it looks for a file "ui.xml" and parses this file (see section 6.6 for more information). The `appName` and `appKey` values are verified with the according values specified in the file. If both match, the application's services are registered and made available to clients. A unique application id is generated and returned in a cookie.

mod_unregister_application

Input:	`appId`
Output:	–

The module unregisters the application with the given id.

mod_request_ui

Input:	`clientId`, `appName`, `schemaName`, `schemaType`, `deviceId`, `services`
Output:	`sessionId`

The module generates and distributes a UI to the client with the given id. If one or more services are provided, user-driven distribution (section 5.3) is chosen. Otherwise, system-driven distribution (section 5.4) is applied. Here, a device profile (linked with the device's id) is needed to assure optimal distribution of services. The `appName`, `schemaName` and `schemaType` parameters guide the module to the schema to use as input for the RELAX NG schema to instance algorithm. If the client is not yet involved in a distribution session, a new session is created. The id of the session is returned in a cookie. The module replies the request with a SEE OTHER response (status code 303) to redirect the client to the UI.

mod_dismiss_ui

Input:	`sessionId`, `service`
Output:	–

The module removes the indicated service from the services assigned to the client. These are stored in its session which is identified by an id. The RELAX NG schema to instance algorithm is invoked (user-driven distribution) with the current services. If the service combination does not result in a satisfiable path, the session is reset, i.e. all services are released and the UI is disposed. This is also achieved if the module is called with "all" as service. The module replies the request with a SEE OTHER response (status code 303) to redirect the client to the UI or to the web interface if the session has been reset.

mod_action

Input:	`sessionId`
Output:	–

The module forwards the message in the request's body to the application involved in the distribution session. The message should be in XML and describe an action triggered by the client. It may be validated against a schema, but the validation is not performed by default for performance reasons.

mod_event

Input:	`appId`, `eventId`
Output:	–

The module forwards the message in the request's body to all clients that are involved

in a distribution session with the given application and that registered to get notified of the concerning event (indicated by the event id). The message should be in XML and describe an event fired by the application. Although the event id is specified in the message, it should also be provided as paramater in order to inform the module of the event id without having to parse the message. The event message may be validated against a schema, but the validation is not performed by default for performance reasons.

mod_last_event

Input:	`sessionId`, `eventId`
Output:	–

The module returns the last event message with the given id that has been sent by the application involved in the session.

mod_ls

Input:	`path`
Output:	`<listing/>`

The module returns a directory listing in XML format. The given path is relative to the webroot.

6.3 The LISTEN method

One of the main challenges we identified is that the HTTP protocol is focused on one-way client-server communication, while our communication model assumes *bidirectional* communication. Figure 6.2 shows that a client can send a message to the IDD (anytime). The IDD's built-in HTTP daemon accepts the connection from the client, receives the message and replies to it. However, the figure also illustrates that the other way round (IDD → client) is all but trivial. This is because the client does not run a server to listen for incoming requests at a certain port. No wonder, because servers always form a potential danger for attacks and clients do not really need them in general. Think for instance of web browsers or UIDL renderers: (basic) server functionality will make them heavier and more vulnerable which may result in a range of security issues and exploits in no time. In the case of standard browsers it is even very difficult – not to say impossible – to listen for incoming connections, without the help of plugins like Java (applets) or ActiveX (which are not supported/available on all platforms). Moreover clients are not expected to process HTTP requests, and neither it is likely that servers would send them. Thereby we can conclude that the HTTP protocol with its default methods and use is not suited for bidirectional communication, given that we consider only *one* server (the IDD). Nonetheless, we still want to stick to the HTTP protocol,

since it is widely used (i.e. each web browser 'speaks' HTTP) and because it is fundamental in the REST approach.

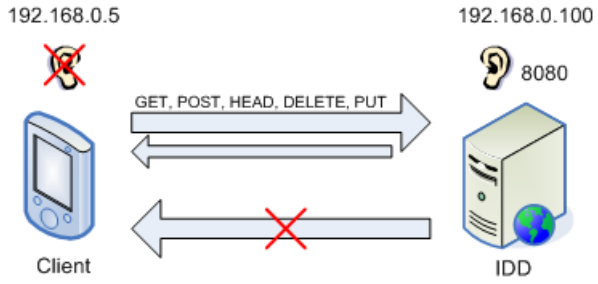


Figure 6.2: Bidirectional communication problem.

Two-way communication between clients and application is a must: if the state of the application changes, the client should be notified and if a client triggers an action, the application should know about it. The IDD mediates between both parties (see section 6.4). Two solutions to realize the IDD → client/application communication are Client Pull and Server Push[3]. Client Pull is to let clients/applications *poll* for actions/events by sending a request to the IDD on regular intervals and let the IDD reply to it with a void or action/event message if one is available. However, polling should be avoided, since it wastes CPU cycles and bandwidth. The IDD should be able to address a client or application directly (i.e. *interrupt* it) and tell it about an action/event. The Server Push technique allows this in a certain extent: a client sends a request, the connection is kept open, and the server can send multiple responses over the connection and trigger the client in time. Although this technique is a very efficient approach, currently it is difficult to use in combination with XMLHttpRequest implementations. It may also be an issue that the IDD does not know if a client/application has actually received the event/action and if it has updated its state while using Server Push.

We introduce the HTTP LISTEN method that allows us to simulate bidirectional communication. The idea is simple: a client (or application) sends a LISTEN request to the IDD, but the request is not answered immediately. On contrary, the IDD waits until it receives an event (e.g. from the application) and *then* replies the LISTEN request with the event message attached as body. Once the client receives the response, it sends another LISTEN request and so on. The use of the LISTEN method is shown in figure 6.3.

However, this only works on condition that the connection (socket) over which the LISTEN request is sent, does *not* timeout. The connection may not be closed, otherwise the server cannot reply on that particular socket and trigger the client. Since we built our own HTTP-based daemon, we can easily avoid timeouts from the server side. Also client-side, we did not discover any problems with timeouts, as browsers or other HTTP supporting clients seem to rely on server side timeouts or provide an option to set an infinite timeout value.

The LISTEN method is a tradeoff between the introduction of a new method and posting to a 'listen-module'. The latter does not require a new method and can easily offer the same

[3] `http://wp.netscape.com/assist/net_sites/pushpull.html`

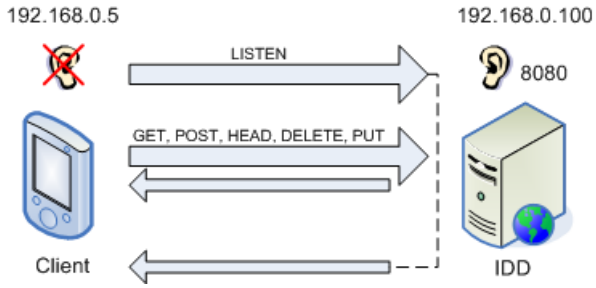


Figure 6.3: Bidirectional communication realized with the LISTEN method.

behavior. Though, traditional web servers respond as fast as possible to POST and GET messages. Therefore it seems unnatural if this general expectation would be broken. The LISTEN method is more intuitive in this context and the HTTP 1.1 specification [15] allows the introduction of new methods.

The LISTEN method can be used by clients as well as by applications. The server automatically detects whether a client or application performed the request by its cookie. Thus, the cookie is obliged; the IDD will ignore the request if no or an invalid cookie is specified. A valid LISTEN call could be:

```
LISTEN / HTTP/1.1
Cookie: appId=d667cc822d3ebe15b174d3fb07105c237806e9ce
```

6.4 Actions and events

Clients and applications can communicate with each other by sending *action* and *event* messages[4]. These messages are typically XML-based, although plain text is also allowed. We assume that messages have the form `<action id="...">...</action>` or `<event id="...">...</event>` where the id identifies the kind of action/event. Their structure is left to the application developer. An application is expected to offer schemas that constrain the structure of valid actions/events. Those schemas can be considered as a 'distributed API', since they provide developers or even agents (e.g. RELAX NG schemas are easy to parse and interpret) with information about what actions are understood by the application and what they should look like (i.e. which tags are implied, which are optional, etc). The same applies for the possible events an application can fire.

If the user performs an action (e.g. she/he presses a button), an action message is sent to the IDD. The IDD forwards this message to the application. The application can trigger an event and send an event message to the IDD with updated state information. The IDD now forwards this event message to all clients that registered interest in this particular event type. Thereby a user action performed on one device can trigger an event that is sent to multiple

[4]Action and event messages are encapsulated inside HTTP messages (as body).

devices. The interface rendered on all of these client devices will be updated according to the new application state. The most recent event message of each kind is stored in a repository, so that new clients entering the interaction space can request them and update their state.

The action/event communication model is exposed in figure 6.4. It actively depends on the LISTEN method to send messages immediately, whenever an action or event occurs. Notice, however, that the model in the figure is a little simplified as it only shows one client and one application. If multiple clients are involved, the IDD will forward an event message over multiple LISTEN streams to all interested clients. In that situation, clients are also likely to trigger actions simultaneously and flood the IDD with messages. Therefore the IDD has a built-in message queue to cache messages and control their flow (figure 6.5). This allows an application to retrieve the actions on its own rhythm, instead of drowning in them.

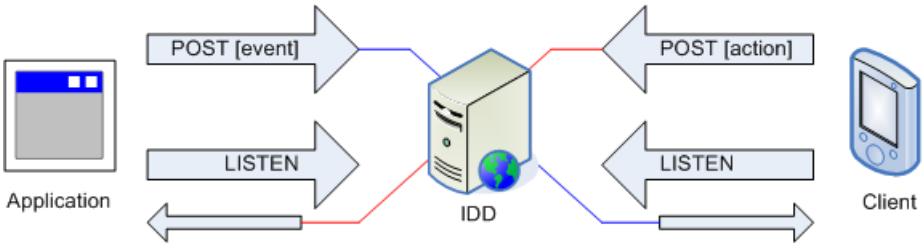


Figure 6.4: Action/event communication model.

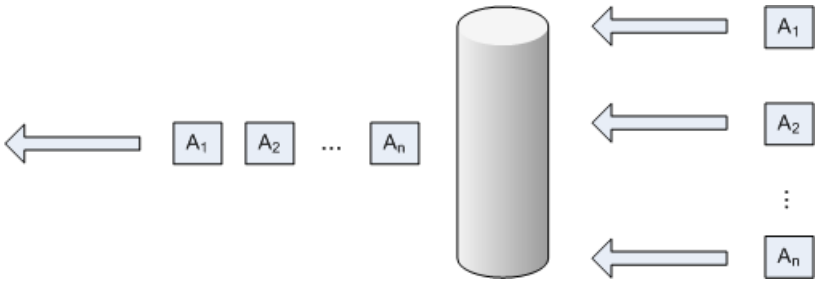


Figure 6.5: Message queue.

Global states

An application may drop some action messages, e.g. if message A_1 (sent by client C_1) causes a change in the application state that disallows the execution of A_2 (sent simultaneously by client C_2), then A_2 should be ignored. But, it should be clear that this situation is far from optimal and may cause conflicts, because often an application will not be able to tell if an action A_i is valid or not. The critical situation arises from an inconsistent application state. If client C_2 was informed of the changed state (e.g. by event E_1) *before* it triggered A_2, its UI would have been updated. For example, a button may have been disabled, disallowing to send A_2. But, message E_1 and A_2 crossed each other and client C_2 received E_1 *after* it sent A_1. The application and client C_2 temporarily faced a different state. This illustrates that

the application and all its distributed UIs *must* have a synchronized global state anytime! This is one of the main difficulties identified in distributed systems. Different mechanisms and algorithms to solve the problem of keeping the global state consistent are outlined in [9]. It is not that difficult to implement such an algorithm in the IDD and guarantee a global consistent state, but it will decrease performance (a lot). Think for example of an interaction space with two clients: one has a fast network connection and one has a slower. If the application transmits an event message, the IDD will forward it and block until it is sure that both clients received the event and updated their state. In particular, the IDD waits for a notification message from each client. The client with the fast connection receives the event first, updates its state, and notifies the IDD. The client with the slow connection does the same, but it takes more time. At this point, the fast client has to wait for the slow client to notify and unblock the IDD and allow further interaction.

In many cases, crossing messages will have no side effects and in these cases, distributed locking is only overhead and becomes redundant. As long as two clients cannot trigger actions that influence the same global variables or dependencies, there is no problem.

6.5 Scenario

This section aims to clarify the architectural concepts discussed in the previous sections via an example scenario. The scenario is visualized in figures 6.6 and 6.7. The first figure shows the registration procedure of an application and a client. Here, the application is a simple music player and the client represents a JavaScript-enabled web browser that supports the XMLHttpRquest object. The registration of the application is the simplest: the application sends a registration request which is answered by the IDD with an OK response. Then the application sends a LISTEN message. For the client, we assume that it uses the IDD's web interface (figure 5.1) to register. The client sends a registration message and the IDD replies with a SEE OTHER message that redirects the client to another web page (with an overview of all applications that registered with the IDD). Next, the user selects the application to interact with and chooses for manual distribution. A message with the selected application, schema and services is posted to the IDD. The IDD now disposes of the required information to generate a UI. All files related with the UI are transferred to the client device. The client renders the UI and transmits a LISTEN request.

Once registered, the client → application interaction can begin as shown in the second figure. The user presses a "play" button and thereby triggers an action message. The IDD receives the message, puts it in the message queue and responds to it with an OK message. Next, the IDD dequeues the message and forwards it to the application as reply on the previously issued LISTEN request. The application immediately sends a new LISTEN request and parses the action message. It notices the play command and starts playing. The application informs the client of its new state by sending an event message, that contains information about the current song. This event message is stored in a repository and forwarded to the client, also in response of a LISTEN request. Finally, the client issues a new LISTEN request,

so it is ready to receive future events. Likewise, the application is prepared for new actions.

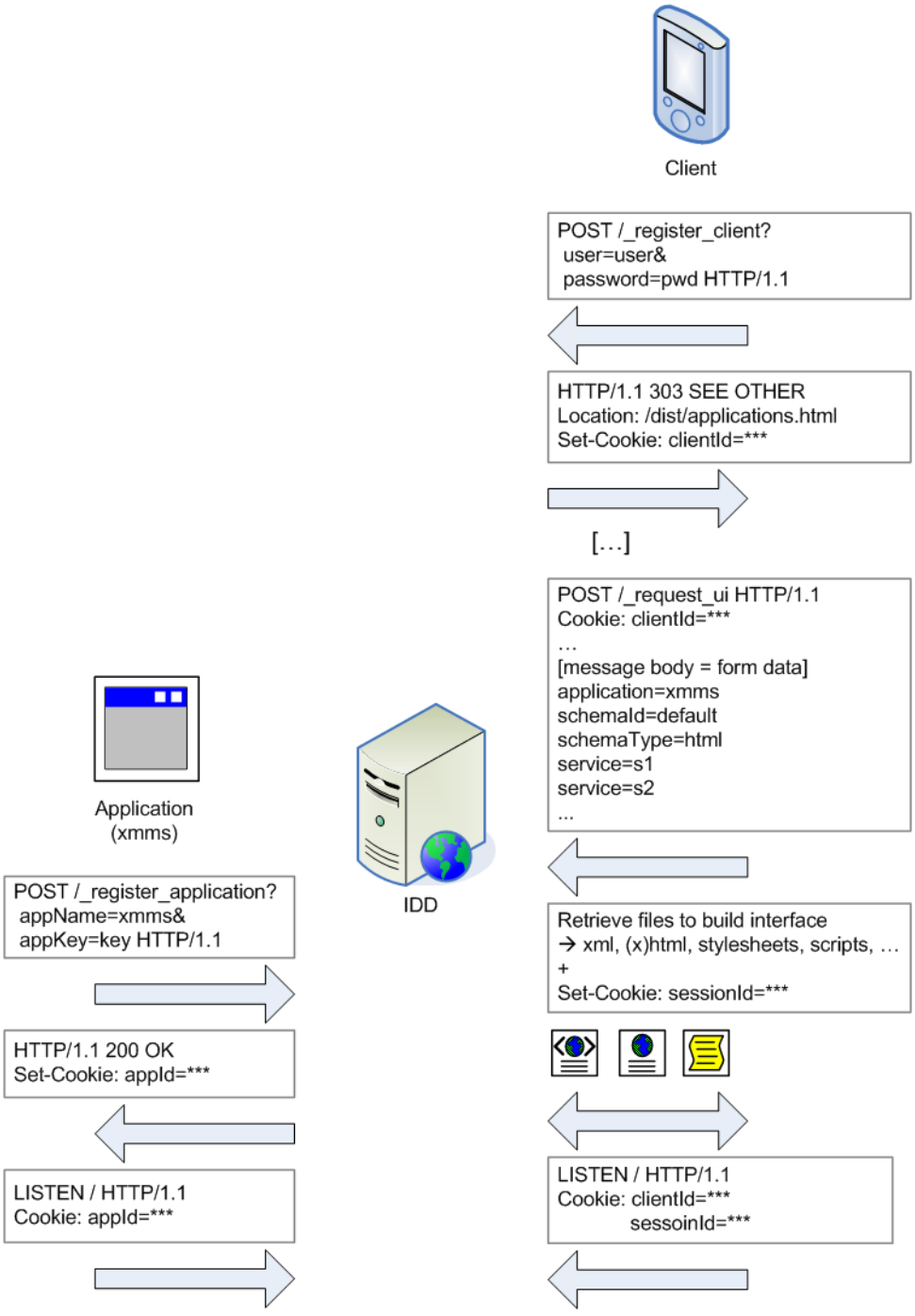


Figure 6.6: Registration procedure.

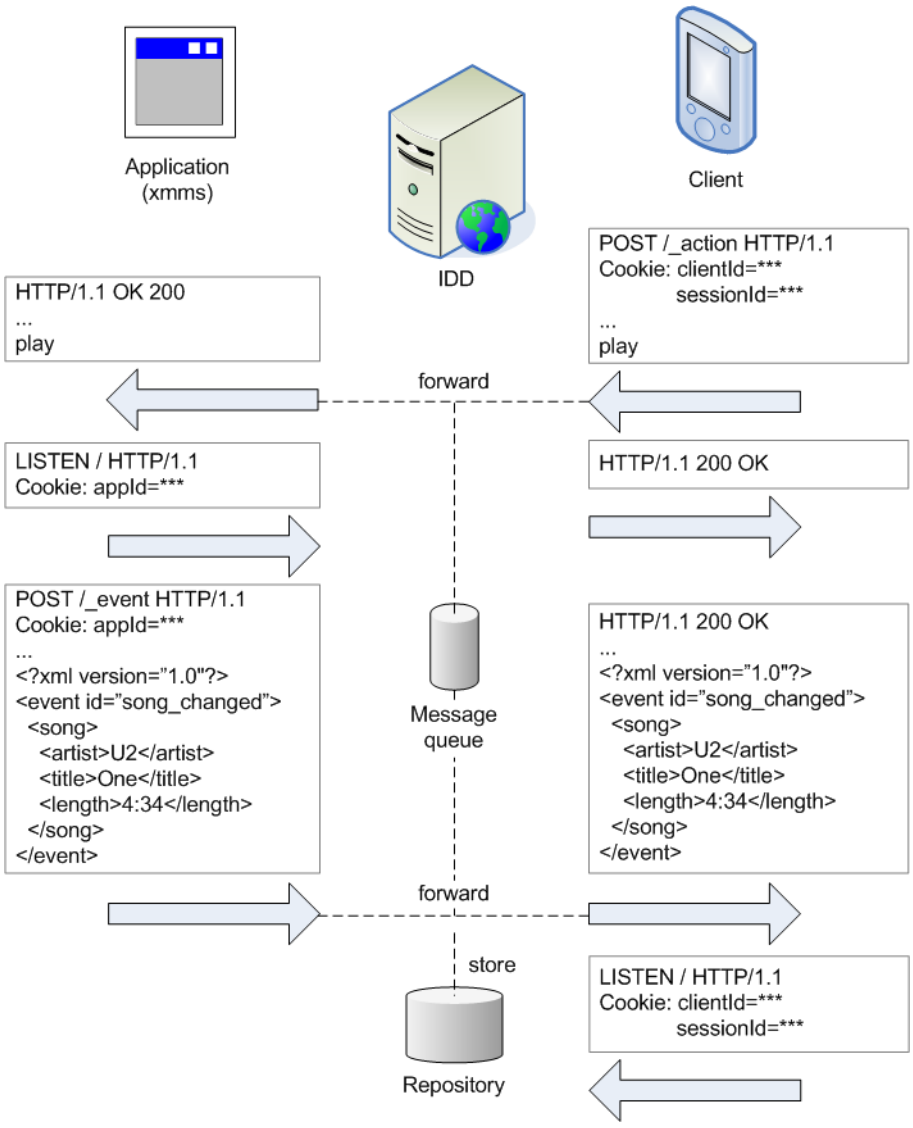
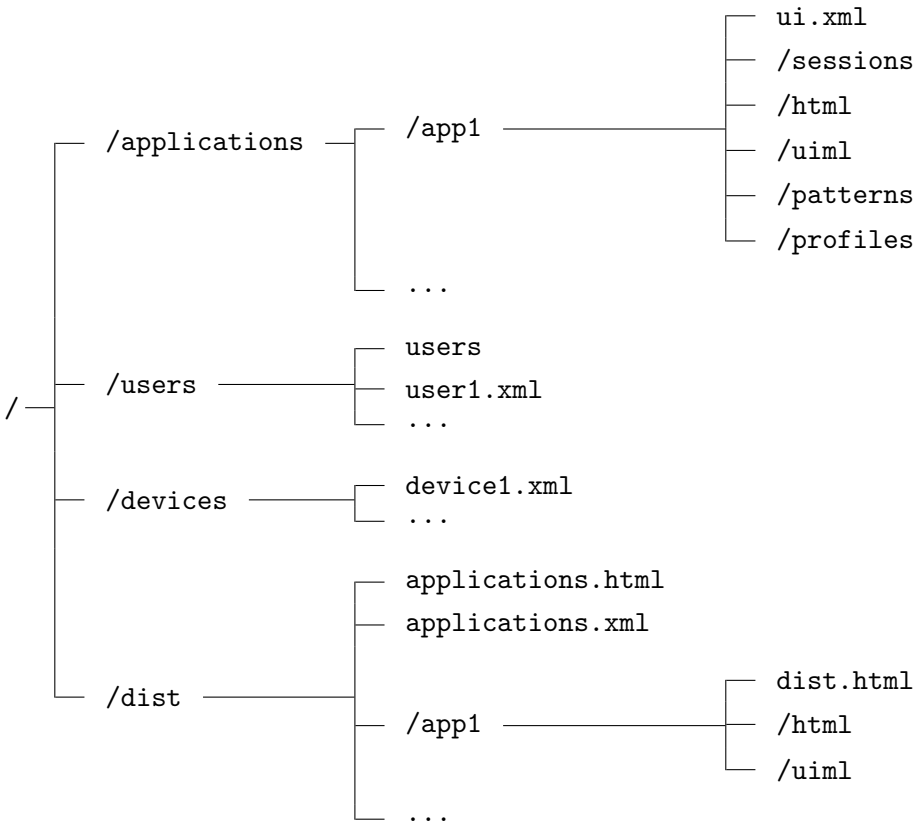


Figure 6.7: Interaction through action and event messages.

6.6 Webroot



`/applications`

The application branch contains a map for each application that pre-registered with the framework. A map contains in the first place the necessary RELAX NG schemas to generate UIs from and related documents (scripts, stylesheets, action/event schemas, ...), but it also holds service profiles (section 5.4.1.2) and a pattern list (section 5.4.2). A session directory is empty by default and is used to cache the distributed UI documents. There is, however, no implied structure on an application map. Instead, a configuration file is used that points to important files and provides the IDD with the necessary settings. This file is named "ui.xml" and is obliged. It is read and parsed when the application registers with the IDD. The next section is devoted to this file.

`ui.xml`

We will take a closer look at the dummy configuration file in figure 6.8, although the semantics of most tags should be clear. In lines 11 – 24, all the services that the application offers are defined. Notice that next to a description, the events to listen for can be specified. A service

can listen for all events (`<AnyEvent/>`) or for particular events (`<event>...</event>`). Often, the latter setting is preferred, as it allows the IDD to forward events only to those clients having a service in their distributed UI that is interested in the event. Next, lines 26 – 29 provide links to the available root RELAX NG schemas. Each schema element has an id and type attribute; their combination is unique and thereby points to a single schema. The schema definitions in the demo configuration file indicate that the IDD will be able to generate/distribute UIs in XHTML and UIML format. The former type of UIs also come in two flavors: "default" and "skin1". Lines 31 – 37 define the service profiles in function of a schema id. Different RELAX NG schemas (skins) can have different profiles associated. Finally, line 40 lists a reference to a pattern list. A pattern list is nothing more than a text file with a number of pattern definitions `1:`P_1, `2:`P_{2a}`;`P_{2b}, ..., one per line. It may be desirable in the future to support multiple pattern lists and thereby allow to choose among various 'distribution schemes'.

`/users`

The user profiles are stored here. User profiles can be uploaded to this map. The folder also contains a "user" file with all valid `username:password` combinations. When a user tries to register with the IDD, its provided username and password are verified against the entries in this file.

`/devices`

The device profiles are stored here. Device profiles can be uploaded to this map.

`/dist`

The dist branch hosts the IDD's web interface. It is updated all the time, e.g. when clients or applications enter/leave the interaction space or if services are distributed. Web agents or non-HTML-based clients may be interested in the "applications.xml" file that contains information about the registered applications and their services and schemas. In fact, that file is transformed to the actual web interface with the help of an XSLT stylesheet [41].

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <application>
4    <!-- general -->
5    <name>demo</name>
6    <key>key</key>
7    <description>Demo application.</description>
8    <version>1.0</version>
9    <group>Multimedia</group>
10
11   <!-- services -->
12   <service name="service1">
13     <description>Service 1.</description>
14     <anyEvent/>
15   </service>
16   <service name="service2">
17     <description>Service 2.</description>
18     <anyEvent/>
19   </service>
20   <service name="service3">
21     <description>Service 3.</description>
22     <event>event1</event>
23     <event>event2</event>
24   </service>
25
26   <!-- schemas -->
27   <schema id="default" type="html">html/default/demo.rng</schema>
28   <schema id="skin1" type="html">html/skin1/demo.rng</schema>
29   <schema id="default" type="uiml">uiml/default/demo.rng</schema>
30
31   <!-- profiles -->
32   <profile schema="default" service="service1">
33     profiles/default/service1.rng
34   </profile>
35   <profile schema="skin1" service="service1">
36     profiles/skin1/service1.rng
37   </profile>
38
39   <!-- pattern lists -->
40   <pattern>patterns/patterns.lst</pattern>
41 </application>
```

Figure 6.8: Configuration file (ui.xml).

6.7 Security

Although security is no primary concern in our framework, we have taken care of a number of security flaws. If a server-based framework is built without *any* notion of possible exploits, the architecture may turn out to be poorly designed when it ever comes to implementing security precautions. Here, [28] was a helpful source to get insight in possible network security issues and solutions.

Secure connections

If messages are sent unencrypted, it figures that it is easier for an attacker to intercept them as if they were encrypted. A (silent) attacker may eavesdrop on a connection or even worse, manipulate data. On a secure line, data is encrypted and thus meaningless for attackers, as long as there are no clues about decryption keys. Connections over the HTTP protocol are typically secured using SSL (Secure Socket Layer). When HTTP is used over SSL, it is called HTTPS (Secure HTTP). Our framework supports both unsafe HTTP connections (default sockets) and secure HTTPS-connections (SSL sockets).

Authentication

Both clients and applications should specify a password (key) in order to register with the IDD. However, the system administrator may allow clients to login anonymously. Of course, sensitive information like passwords are best sent over a secure connection.

Session Management

If an application receives an action message, it must be sure that this message comes from a valid client in the interaction space. Also, if a client receives an event message, it must accept that it comes from the expected application. In both situations, the IDD is the central entity that delivers the messages, i.e. forwards them. The IDD should verify all incoming messages to be sure they are not sent by a third party (spoofing). It does so by comparing clientIds, appIds and sessionIds (accompanying the message in a cookie) with an internal list of active ids. Notice that these (random) ids are created upon registration or at the start of a distributed session and thereby act as temporary keys to allow the delivery of messages. If an id does not match or is not specified, the message is rejected. The ids are also used to identify a client/application and look up the (final) destination(s) of the message.

Read and write permissions

Filesystems often allow to set permissions and restrict read/write access to certain files or directories. The same idea is applicable to web servers that require authentication for certain

files, or – if they e.g. support the PUT method – only allow to upload files to specific directories. In the context of the IDD, read access should be restricted as we do not want users to view an application's configuration file or a list of users/passwords, given that these files (currently) reside in the webroot. Off course, write access should also be restricted to well-defined directories, e.g. to upload profiles. However, we did not elaborate on this and just rely on filesystem permissions.

Chapter 7

Deployment

7.1 Introduction

A framework may offer powerful technology, but as long as this technology cannot be joined with applications people are used to work with, it has little chance to become a success. If developers/users see hail in the technology, it is important that they are able to use and incorporate it with minimal effort. We tried to achieve this goal in our framework by means of high-level APIs that allow the developer to focus on the design of a distributable UI instead of on the framework itself. The developer does not need to learn a framework-specific language, but she/he can use an XML-based UIDL language (HTML, UIML, ...) of choice. In the next sections, we explain how we *extended* an audio player (XMMS) and an image viewer (JShow) with the ability to offer distributed service UIs. We also outline what is to be done to integrate the framework with an (advanced) drawing program (the GIMP). Migratable UIs create new opportunities and one of them is to replace traditional appliance remotes. We discuss a PDA/UI combination to control a home multimedia system and domotica software.

7.2 Native applications

7.2.1 XMMS

One of the first objectives was to connect a simple yet well-known application with the IDD and allow it to be controlled by means of a distributed UI. The choice fell on XMMS[1], a popular linux audio player. Important to notice is that XMMS already has a (even skinnable) GUI. The application logic is no stripped terminal program running on some server, but just the regular application as it is used and known by people. In fact, many solutions exist to control XMMS from various devices. They broadly fall apart in two categories. On the one hand there is the web server approach: a web interface is hosted on a web server and with

[1] `http://www.xmms.org`

the help of technologies such as PHP, users are able to control the music player from a web browser. On the other hand there is the client and server program solution: a server program (perhaps a plugin) is installed on the PC running XMMS and a (mostly platform-dependent) client program is installed on the device to control XMMS from. The latter setup requires a lot of work as the emphasis is on the application and the target platform to control it from. Sure the remote UI (client program) is probably visually appealing and works just fine, but it is no generic solution that can be ported with little effort to other applications and platforms. The web server way is more interesting in this context, since most devices have a web browser on board and thus are able to render compliant web sites. However, a web interface is not closely integrated with the operating system and has different looks and behavior than the UI of native applications. This may not be desirable. Another issue is the server-side scripting and programming work involved to make things work. The use of the IDD overcomes these issues and adds some extra advantages:

- The IDD acts as **middleware** between XMMS and the client device(s) to control it. The REST API offers the necessary functions to have client devices communicate with the application logic. Thereby it relieves the UI designer of many programmatic squirms.
- A UI can be distributed (and even duplicated) among different devices. The UI can be offered in **several formats** (HTML, UIML, ...) and visualized by a renderer of choice. This can be a web browser, but also a UIDL renderer that uses the platform's default UI toolkit to render the UI.
- There is no need for XMMS to be installed on the same PC as the IDD.

We designed two simple shell scripts in order to pass actions to XMMS and forward events to the IDD. We could have written a plugin for XMMS as well, but the scripts are easier to adapt since they do not need to be compiled or installed. In fact, the scripts rely on an existing XMMS plugin: `xmmsctrl`. This plugin allows to control XMMS from the command line. It accepts a bunch of parameters such as play/stop/... and it also gives information on the current song playing. The scripts use the CURL library[2] to send and receive HTTP messages. One script periodically polls XMMS for changes and the other listens for incoming actions by using the LISTEN method (section 6.3), but none of them run a server. Action commands (e.g. play) are just sent in plain text in the body of an HTTP message. The same applies for event data such as name and artist of the current song or an entire playlist.

Then a web interface for XMMS was built that consists of three parts, like described in section 4.2. Designing the parts is the same as designing a regular XHTML-compliant web page. JavaScript and the XMLHttpRequest object are used for background communication with the IDD, thereby supporting the AJAX web application model (section 2.5). A single JavaScript source file provides the necessary functions to interact with the IDD. This file is included in the web interface so that its functions can be invoked. Notice that the source file can be reused since it has nothing to do with XMMS specifically! It can be considered as a

[2] `http://curl.haxx.se/`

binding between a web browser and the IDD. Once the parts (XHTML code) are ready, they should be converted to RELAX NG schemas. This is very simple and the process can easily be automated. Finally, a "ui.xml" file is needed before merging the service interfaces (main, playlist, settings) into the IDD's webroot.

Figure 7.1 gives an idea of the result: a distributed UI for XMMS is split among two clients. The first client has the main and settings service, while the second disposes of the playlist service. Notice that JavaScript is used to add local functionality to the UI for the playlist service. Once the playlist (we refer to the actual list of songs loaded in XMMS here, not the service) is received, it is possible to filter songs on the fly *without* interacting with the IDD or XMMS. They are filtered client-side, and if a song is clicked, the track id is sent to the IDD and forwarded to XMMS that starts playing the song.

7.2.2 JShow

A second supported application is JShow, a simple image viewer we wrote. It can be used as a stand-alone application (just like any other image viewer), but it can also be controlled remotely. Its services become available to clients as soon as JShow is connected with the IDD (remote control enabled). These services are:

- browser: browse through the file system and open a directory with images.
- picture: shows the full sized image, possibly in a scrollable canvas if it is too large to fit.
- scroll: scroll left/right/up/down.
- thumbnail: shows a thumbnail version of the current image.
- view: jump to the first/previous/next/last image.
- zoom: zoom in/out on the image.

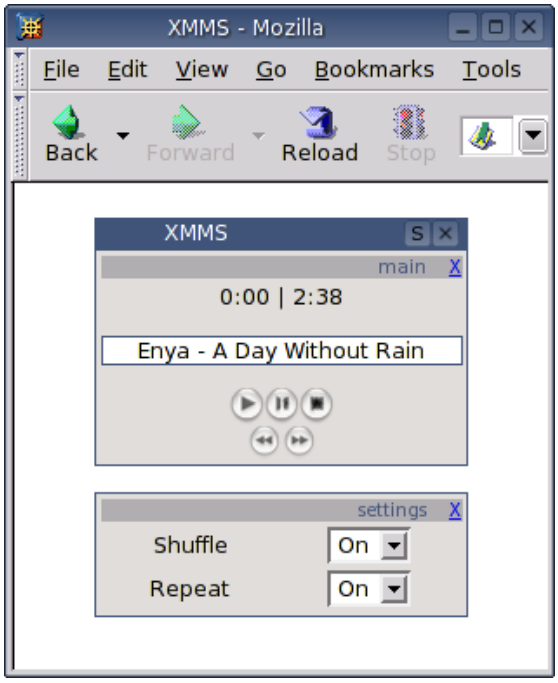
We designed two web interface themes for these services, likewise as described in the previous section. The first (default) supports all services and is shown in figure 7.2(a). The main RELAX NG schema that constrains the structure of this web interface, defines that the thumbnail service may not be distributed together with the picture service. It also says that scroll and zoom parts should appear next to each other, while the other parts should be put below each other. The other theme (mac) is limited to the thumbnail and view service; it is shown in figure 7.2(b). The mac theme is optimized for devices with a small screen. We also designed a simple UIML interface; a screenshot is shown in figure 7.2(c). Both the web interface as the UIML interface can be used concurrently and services can migrate from the one to the other. Figure 7.3 shows a UI for JShow rendered in Pocket Internet Explorer on a PDA (Pocket PC), and figure 7.4 shows an interaction space with different client devices involved.

The communication between JShow and the IDD happens through a 'connector' API. This is a simple Java library independent of the application that acts as a bridge between the application and the IDD, just like the XMLHttpRequest driver script discussed in the previous section is a binding between the IDD and a web browser. The abstraction is outlined in figure 7.5. Thanks to the high-level APIs, all transactions are handled transparent to the developer.

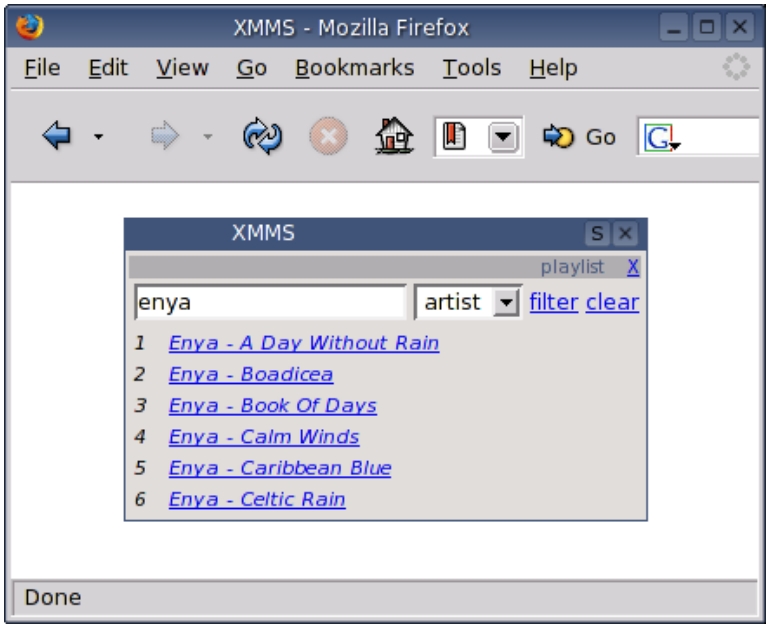
7.2.3 The GIMP

We will shortly outline what is to be done to get drawing programs such as the GIMP[3] work with the framework. A useful scenario might be the following: a tablet PC exposes the image canvas full screen so that the user can directly 'draw' on the screen of the device. A second device, a PDA, shows the toolbox and allows to change from pencil to color picker or any other tool. In this scenario (figure 7.6), the GIMP itself can run on the tablet PC and show its canvas maximized. The GIMP should be connected with the IDD and have a distributable UI for the toolbox that can talk with the GIMP core. Therefore a plugin is required that is able to parse and interpret XML-based action messages and translate them to native function calls. Fortunately, the source code for such a plugin can be kept very simple, since there are XML toolkits and HTTP libraries around for practically every operating system and programming language that take most of the work out of the hands of the developer. It can be kept even simpler as soon as there is an independent, reusable library (API) for the C(++) programming language, binding an arbitrary C(++) application with the IDD, analogous to the Java-based connector used by JShow (figure 7.5). Writing this API is no big deal either. The required plugin thus basically consists of dispatch code and will count only a few lines. The UI for the GIMP toolbox (service) can be created similarly as the ones for XMMS and JShow. Likewise, UIs for other GIMP components (e.g. layers dialog) can be created and distributed to different devices in the environment. Again, we want to emphasize that there is no need to stick to a web interface.

[3] `http://www.gimp.org/`



(a)



(b)

Figure 7.1: Distributed web interface for XMMS, main and settings service rendered in Mozilla (a) and playlist service rendered in FireFox (b).

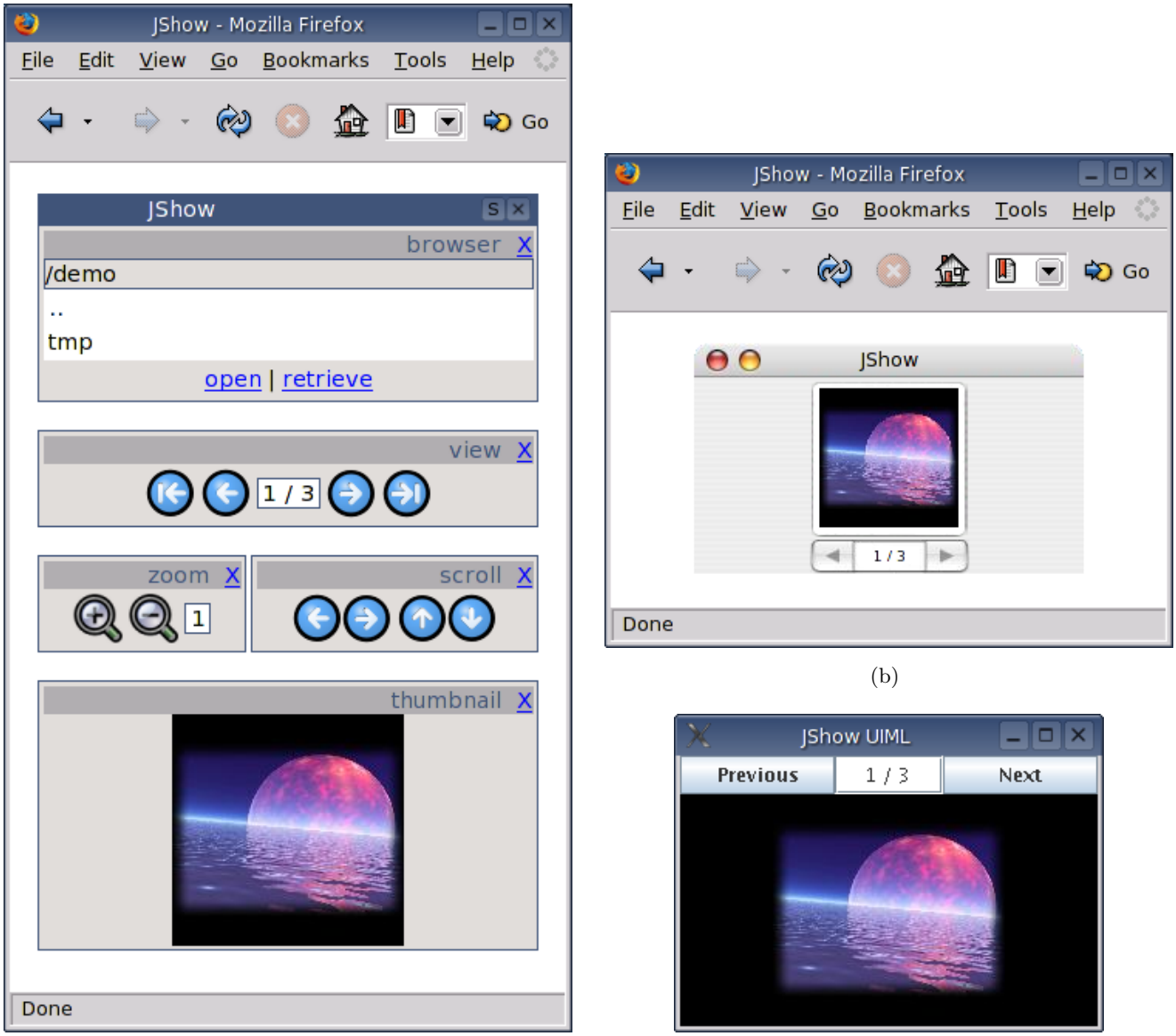


Figure 7.2: Distributed web interface for JShow, default (a) and mac (b) theme rendered in the FireFox browser and distributed UIML interface rendered using a Java Renderer from Harmonia.

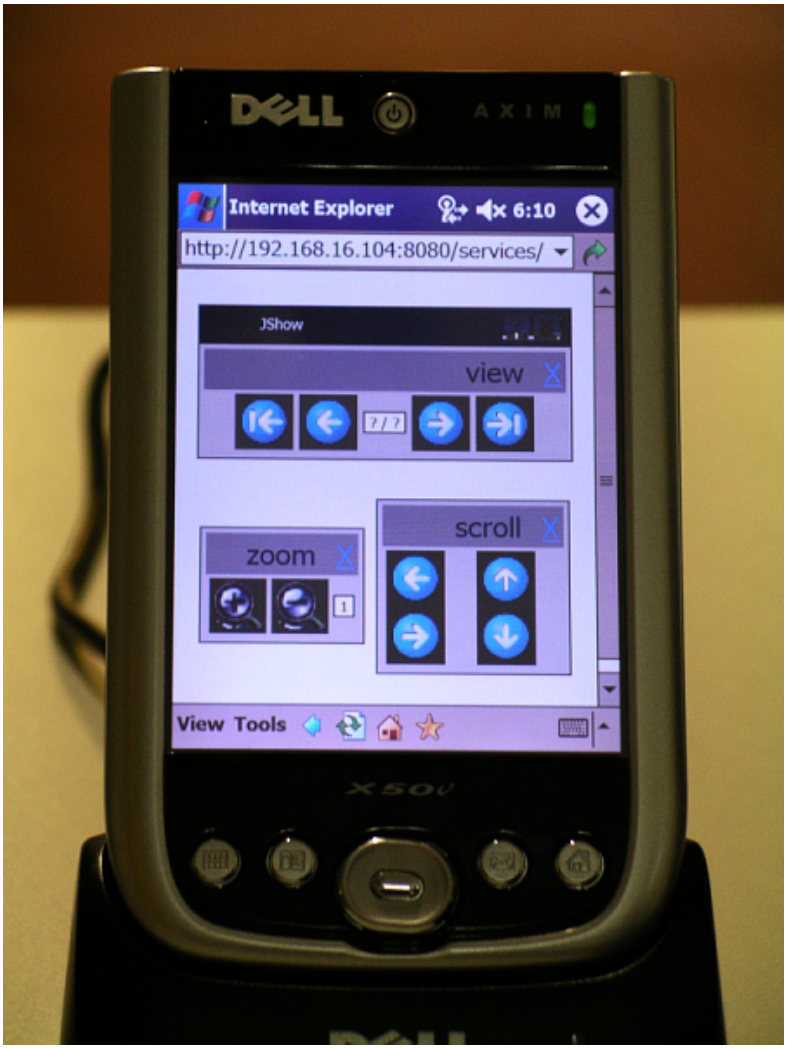


Figure 7.3: Distributed web interface for JShow, rendered in Pocket Internet Explorer on a PDA.

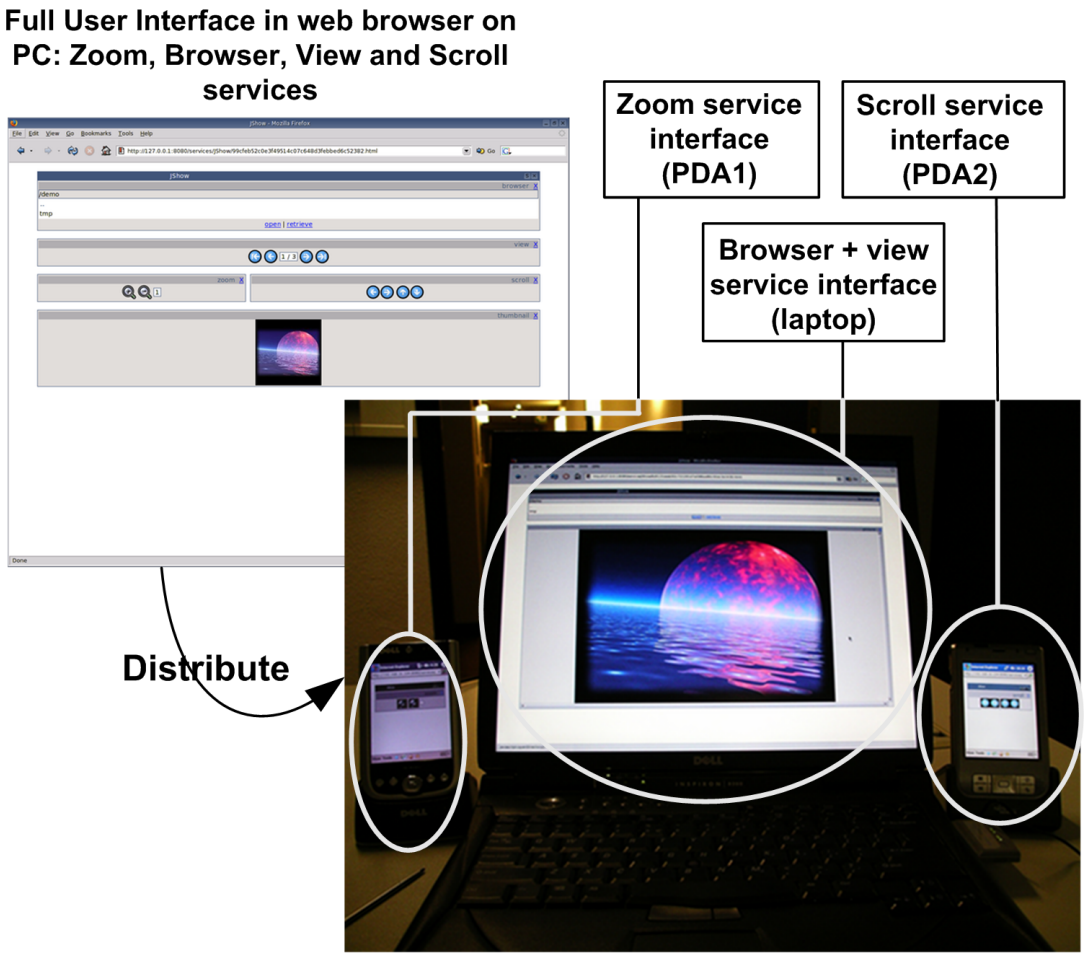


Figure 7.4: Scenario in which a web interface for JShow is spread over two PDAs and a laptop.

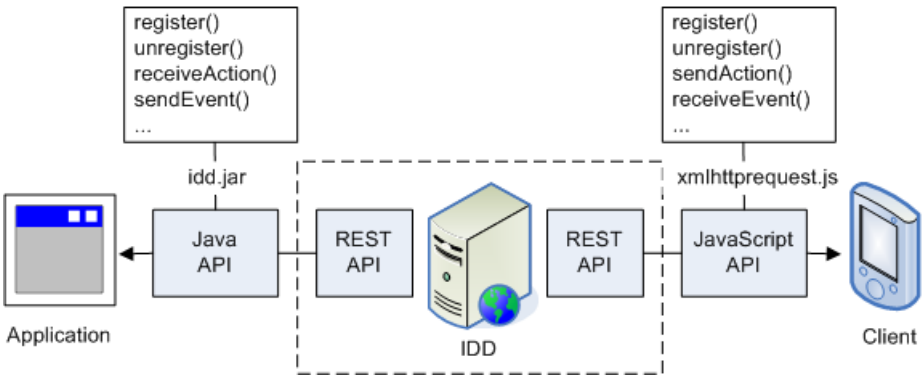


Figure 7.5: Abstraction of APIs to ease the application ↔ IDD and client ↔ IDD communication to developers.

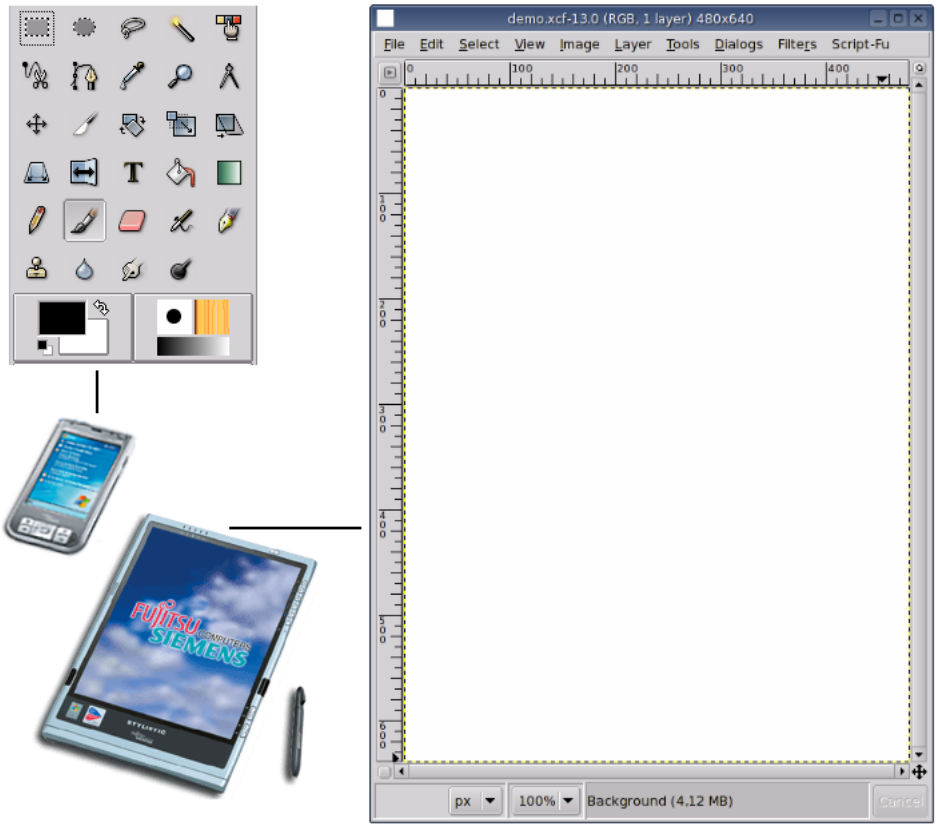


Figure 7.6: Scenario in which the GIMP toolbox is migrated to a PDA and the drawing canvas is loaded full screen on a tablet PC.

7.3 Multimedia Center and Domotica

Lately, there is a growing interest in silent and stylish PCs integrated in the living room. These small PCs (called 'barebones') can replace the VCR, DVD recorder, tuner, CD player and other appliances. A multimedia barebone may be connected with a large LCD panel and 7.1 surround sound set. A music collection can be stored right on its hard disk and played through the surround speakers. TV programs can be recorded on the hard disk and shared over the home network. Nowadays, multimedia barebones are typically delivered with Windows XP Media Center Edition and a basic IR (infrared) or RF (radio frequency) remote. In our experience, this remote spoils the beauty of the barebone setup. A multimedia center offers so many options and a traditional remote cannot possibly deal with them all on its own. The cartoon in figure 7.7 captures the abundance of features in a nice way. The fact that a remote can only have a limited number of buttons is solved by letting it operate a GUI (running on the barebone) by means of arrow/ok/cancel buttons. Hence, the remote is to be used in conjunction with the LCD panel on which the GUI is shown. However, plastic or rubber feeling buttons are not the most comfortable way to browse through menus and execute actions.

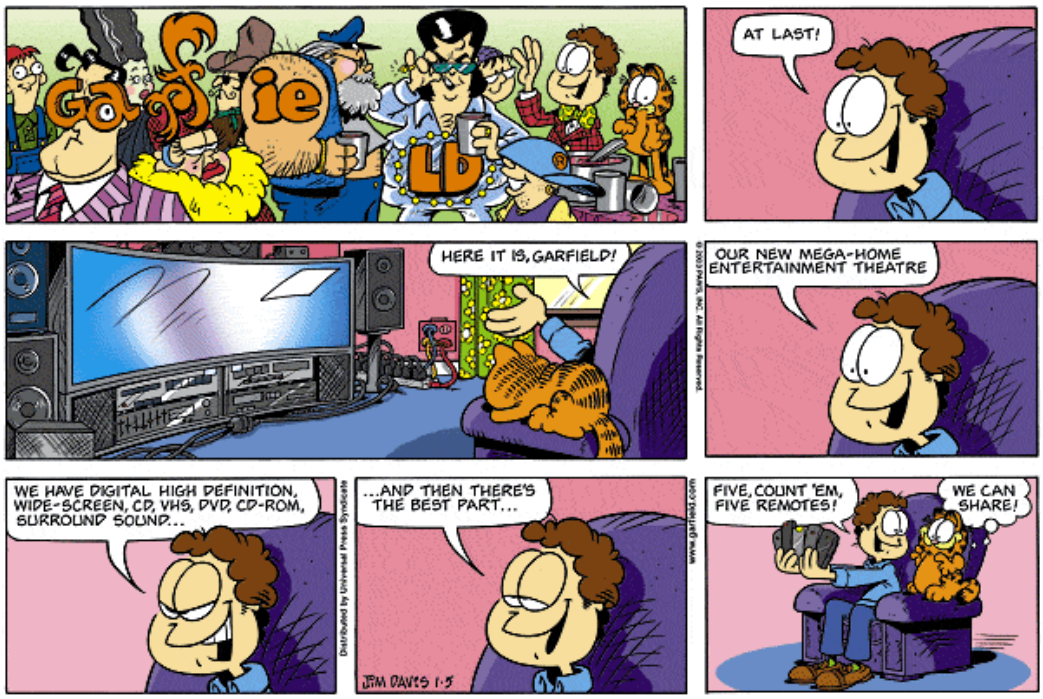


Figure 7.7: Garfield comic, copyright by Jim Davis.

An option is to replace the remote by a PDA to control the home entertainment. This can be achieved by installing the IDD somewhere, preferrably on a PC that acts as *residential gateway*, and registering the software that runs on the barebone with the IDD. The idea is to let the user request a distributed UI for the service she/he wants to use. For example, if one intends to program a recording, *only* a UI for that specific task is needed. In the case of

a PDA, buttons can be pressed by tapping on the screen. Visual feedback is given directly on the PDA display and/or on the LCD panel. The distributed UI also allows to select songs or program recordings from *anywhere* in the house (WLAN) or even in the world (WLAN + Internet). PDA-like touch screen devices are also found in the Philips Pronto Series[4]. These devices are able to transmit and record IR and/or RF codes and offer a local, customizable UI to the user. The UI resides on the device itself and is not distributed, but the way of interacting with appliances is highly comparable.

There is also an opportunity to exploit the IDD for domotica purposes and connect it with a domotica software system. This way, the IDD can offer service UIs to switch on/off or dim the lights, set the temperature, open/close the shutters, ... If these services are available over the Internet, security is an important concern as we do not want strangers to switch off the central heating or set the thermostat to 40 degrees.

[4] `http://www.pronto.philips.com/`

Chapter 8

Conclusions

8.1 Summary of Results

To overcome the diversity in mobile devices and design portable GUIs, User Interface Description Languages (UIDLs) proof to be very useful. The idea is to describe a GUI in high-level terms and render the actual GUI from this description by means of a native widget set available on the target device. A similar approach is to use the matured (X)HTML markup language to design web interfaces, possibly in combination with stylesheets (CSS) and scripts (JavaScript). Although developers of web interfaces have to deal with a limited set of widgets, they can profit from the fact that practically every platform supports a web browser able to visualize the interface. It has little sense to describe a GUI in a powerful UIDL if there are no maintained renderers available for that UIDL.

Since these high-level descriptions are fairly portable, they are a good base for our primary objective: distributable interfaces. We call a UI distributable if it can migrate as a whole to a single device or in predefined parts to multiple devices. An important observation in this context is that users are likely to use only a few specific services an application offers. In this case, only parts of a GUI are needed and distributed to the user's device. A smart distribution of those parts that are needed to fulfill a certain task is a must, given the limited screen size of most portable devices. An XML-schema language (RELAX NG) is applied to describe the structure, constraints and types of a set of service UIs. We distinguish between user-driven, system-driven and continuous distribution. The main difference between the former two is the actor – the user or the system – that decides what services (UI parts) are distributed to what devices in the interaction space. Continuous distribution involves the redistribution of a GUI if clients enter or leave the interaction space, yet this is a topic for future research.

In a distributed environment, communication between devices is of considerable importance. Wireless ethernet and in particular IP-based networks are well suited to interconnect hosts in the interaction space. One of the advantages of the IP protocol is its direct link with the Internet, which stands guarantee for a wide coverage. An attractive higher-level protocol that runs on top of the IP/TCP protocol stack is HTTP, commonly used in combination with

the web or related services. RPC-based middleware systems often piggyback on the HTTP protocol and are of particular interest in service-oriented environments. Especially the REST 'vision' attracts the attention because of its simplicity.

In the framework presented, we combine schema-driven UI generation/distribution with REST. The framework is successfully applied to control a native audio player and image viewer by means of a dynamic distributed interface, spread over one or more devices (laptop, PDA, ...). It is built so that developers can extend applications with minimal effort and focus on the design of a distributable GUI without the need to learn a framework-specific language or worry about distributed communication.

8.2 Concluding Remarks

In advance to this text, a paper [32] has been accepted by the International Conference on Web Engineering[1] that discusses some of the major foundations outlined here. The paper focuses on web interfaces (XHTML), but the presented framework can deal with any XML-based description language. Some words from the reviewers of the paper:

> The distribution of a web page, along with its services, is an interesting topic that is not yet completely solved. The distribution problems makes really sense when different computing platforms are used, perhaps with different sets of constraints. The system presented in the paper takes into account some constraints (mainly, the screen resolution) of the computing platforms available to distribute the web page. The paper clearly shows the problem posed by web distribution and demonstrates that the authors have handled the problem very straightforwardly. A very nice differentiation is made between personal interaction space and collaborative interaction space.

Movies, screenshots and some full sized images of the system in use are available on the following URLs:

- `http://research.edm.luc.ac.be/cvandervelpen/research/icwe2005/`
- `http://lumumba.luc.ac.be/gvanderh/projects/thesis/`

[1] `http://www.icwe2005.org`

Bibliography

[1] Marc Abrams, Constantinos Phanouriou, Alan L. Batongbacal, Stephen M. Williams, and Jonathan E. Shuster. UIML: An Appliance-Independent XML User Interface Language. *Computer Networks*, 31(11-16):1695–1708, 1999.

[2] Renata Bandelloni and Fabio Paternò. Flexible Interface Migration. In *Proceedings of Intelligent User Interface 2004 (IUI 04)*, pages 148–155, 2004.

[3] Tim Berners-Lee, James Hendler, and Ora Lassila. *The Semantic Web*. Scientific American, May 2001.

[4] Philip A. Bernstein. Middleware: a Model for Distributed System Services. *Communications of the ACM*, 39(2):86–98, 1996.

[5] Silvia Berti, Francesco Correanim, Fabio Paternò, and Carmen Santoro. The TERESA XML Language for the Description of Interactive Systems at Multiple Abstraction Levels. In *1st Annual International Conference on Mobile and Ubiquitous Systems 2004 (MOBIQUITOUS 04)*, pages 103–110, 2004.

[6] Galle Calvary, Joëlle Coutaz, Olfa Dâassi, Lionel Balme, and Alexandre Demeure. Towards a new generation of widgets for supporting software plasticity: the "comet". In *The 9th IFIP Working Conference on Engineering for Human-Computer Interaction, Jointly with The 11th International Workshop on Design, Specification and Verification of Interactive Systems*, 2004.

[7] James Clarck and Makoto Murata. *RELAX NG Specification*. World Wide Web, `http://www.relaxng.org/spec-20011203.html`, 2001.

[8] Karin Coninx, Kris Luyten, Chris Vandervelpen, Jan Van den Bergh, and Bert Creemers. Dygimes: Dynamically Generating Interfaces for Mobile Computing Devices and Embedded Systems. In *Mobile HCI*, pages 256–270, 2003.

[9] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed systems: Concepts and Design*. Addison-Wesley, 3rd edition, 2001.

[10] Jacob Eisenstein, Jean Vanderdonckt, and Angel R. Puerta. Applying model-based techniques to the development of UIs for mobile computers. In *Intelligent User Interfaces*, pages 69–76, 2001.

[11] Kent Fitch. *Schema Driven User Interface Generation*. `http://ausweb.scu.edu.au/aw02/papers/refereed/fitch/paper.html`, 2002.

[12] Krzysztof Gajos and Daniel S. Weld. SUPPLE: Automatically Generating User Interfaces. In *IUI*, Funchal, Portugal, 2004.

[13] Donatien Grolaux, Peter Van Roy, and Jean Vanderdonckt. Migratable User Interfaces: Beyond Migratory Interfaces. In *Sattelite workshop of Advanced Visual Interfaces (AVI 2004)*, pages 422–430, 2004.

[14] The Internet Engineering Task Force (IETF). *HTTP State Management Mechanism*. `http://www.ietf.org/rfc/rfc2109.txt`, February 1997.

[15] The Internet Engineering Task Force (IETF). *Hypertext Transfer Protocol – HTTP/1.1*. `http://www.ietf.org/rfc/rfc2616.txt`, June 1999.

[16] ECMA International. *ECMAScript Language Specification*. `http://www.ecma-international.org/publications/standards/Ecma-262.htm`, 3rd edition, December 1999.

[17] Choonhwa Lee and Sumi Helal. Protocols for Service Discovery in Dynamic and Mobile Networks. *International Journal of Computer Research*, 11(1):1–12, 2002.

[18] Kris Luyten and Karin Coninx. UIML.NET: An open UIML renderer for the .NET framework. Technical report, Limburgs Universitair Centrum, 2004.

[19] Guido Menkhaus and Sebastian Fischmeister. Dialog model clustering for user interface adaptation. In *Web Engineering, Proceedings of the International Conference on Web Engineering 2003 (ICWE 03)*, volume 2722 of *LNCS*, pages 194–203. Springer Verlag, 2003.

[20] Giulo Mori, Fabio Paternò, and Carmen Santoro. Design and development of multidevice user interface through multiple logical descriptions. *Transactions on Software Engineering*, 30(8), 2004.

[21] Brad A. Myers. Using handhelds and PCs together. *Commun. ACM*, 44(11):34–41, 2001.

[22] Netscape. *Persistent Client State HTTP Cookies*. `http://wp.netscape.com/newsref/std/cookie_spec.html`, 1999.

[23] Jeffrey Nichols, Brad A. Myers, Michael Higgins, Joe Hughes, Thomas K. Harris, Roni Rosenfeld, and Mathilde Pignol. Generating Remote Control Interfaces for Complex Appliances. CHI Letters: ACM Symposium on User Interface Software and Technology. In *UIST*, Paris, France, October 2002.

[24] Shankar Ponnekanti, Brian Lee, Armando Fox, Pat Hanrahan, and Terry Winograd. ICrafter: A Service Framework for Ubiquitous Computing Environments. In *Proceedings of the 3rd International Conference on Ubiquitous Computing (UBICOMP 01)*, pages 56–75. Springer-Verlag, 2001.

[25] Shankar Ponnekanti, Luis Alberto Robles, and Armando Fox. User Interfaces for Network Services: What, from Where, and How.

[26] Nathalie Souchon and Jean Vanderdonckt. A Review of XML-Compliant User Interface Description Languages. In Joaquim A. Jorge, Nuno Jardim Nunes, and João Falcão e Cunha, editors, *DSV-IS*, volume 2844. Springer, 2003.

[27] Pedro Szekely. Retrospective and Challenges for Model-Based Interface Development. In F. Bodart and J. Vanderdonckt, editors, *Design, Specification and Verification of Interactive Systems*, pages 1–27, Wien, 1996. Springer-Verlag.

[28] Andrew S. Tanenbaum. *Computer Networks*. Prentice-Hall, 4th edition, 2003.

[29] Eric van der Vlist. *RELAX NG*. O'Reilly and Associates, 2003.

[30] Sander van der Wal. Designing and building portable UIs for Symbian OS: Using multiple controllers. Technical report, mBrain Software, March 2004.

[31] Jean Vanderdonckt and François Bodart. Encapsulating knowledge for intelligent automatic interaction objects selection. In *Proceedings of the SIGCHI conference on Human factors in computing systems 1993 (CHI 93)*, pages 424–429. ACM Press, 1993.

[32] Chris Vandervelpen, Geert Vanderhulst, Kris Luyten, and Karin Coninx. Light-weight Distributed Web Interfaces: Preparing the Web for Heterogeneous Environments. 2005.

[33] World Wide Web Consortium (W3C). *Cascading Style Sheets (CSS)*. `http://www.w3.org/Style/CSS/`.

[34] World Wide Web Consortium (W3C). *Extensible Markup Language (XML)*. `http://www.w3.org/XML/`.

[35] World Wide Web Consortium (W3C). *RDF Vocabulary Description Language 1.0: RDF Schema*. `http://www.w3.org/TR/rdf-schema/`.

[36] World Wide Web Consortium (W3C). *Resource Description Framework (RDF)*. `http://www.w3.org/RDF/`.

[37] World Wide Web Consortium (W3C). *Simple Object Access Protocol (SOAP)*. `http://www.w3.org/TR/soap/`.

[38] World Wide Web Consortium (W3C). *Web Ontology Language (OWL)*. `http://www.w3.org/2004/OWL/`.

[39] World Wide Web Consortium (W3C). *XForms - The Next Generation of Web Forms*. `http://www.w3.org/MarkUp/Forms/`.

[40] World Wide Web Consortium (W3C). *XML Schema*. `http://www.w3.org/XML/Schema`.

[41] World Wide Web Consortium (W3C). *XSL Transformations (XSLT) Version 2.0*. `http://www.w3.org/TR/xslt20/`.

[42] World Wide Web Consortium (W3C). *The Extensible HyperText Markup Language (XHTML).* `http://www.w3.org/TR/xhtml1/`, January 2000.

[43] World Wide Web Consortium (W3C). *Composite Capability/Preference Profiles (CC/PP): Structure and Vocabularies 1.0.* `http://www.w3.org/TR/CCPP-struct-vocab/`, 2004.

[44] World Wide Web Consortium (W3C). *Document Object Model (DOM).* `http://www.w3.org/TR/DOM-Level-3-Core`, April 2004.

[45] World Wide Web Consortium (W3C). *Document Object Model (DOM) Level 3 Load and Save Specification.* `http://www.w3.org/TR/DOM-Level-3-LS/`, 2004.