Chapter 8
Hypermedia-Driven Framework for Scalable and Adaptive Application Sharing

Vlad Stirbu and Juha Savolainen

Abstract This chapter describes our experiences designing a solution for scalable and adaptive sharing of desktop and mobile applications, using a lightweight network-based system compliant with the REST architectural style. The system delivers consistency of the rendered user interfaces with the state of the application logic using a stateless networking substrate. We describe the architecture focusing on how to model the user interfaces as a set of web resources. Then, we present the prototype that implements the functionality as an extension of the Qt framework, which works with different Qt-based user interface toolkits. Finally, we present a multi-display and multi-user Texas Hold'em application that shows how the system is used in practice.

Introduction

Sharing the user interface of an application allow users to control applications from remote computers. Modern mobile devices with sophisticated capabilities challenged the traditional role of desktop or laptop computers as users' preferred devices. They now expect to access applications anytime and anywhere while the usage experience is optimized for the particular device.

Traditional approaches on sharing the user interface relied on transferring the content of the framebuffer or the drawing commands from the device running the application to the device rendering the user interface. These techniques do not provide appropriate results for mobile devices or consumer electronics, which typically have smaller displays and/or different interaction metaphors. Therefore, to improve the user experience, the shared user interfaces have to be adapted to the rendering device look and feel.

V. Stirbu (✉)
Nokia Research Center, Visiokatu 1, Tampere 33720, Finland
e-mail: vlad.stirbu@nokia.com

Motivation

Traditionally, application sharing relied on remote user interface protocols that export, the content of the framebuffer (e.g. like VNC), or the drawing commands for the Graphic Device Interface (GDI) (e.g. like X Windows system). With these approaches the user interface is rendered on the remote device as instructed by the server with little or no possibility of customization.

Application sharing emerged when direct access to computing devices offered by mainframes and servers was not easily available. However, modern applications are developed using sophisticated frameworks that typically use the Model View Controller (MVC) (Krasner and Pope 1988) design pattern and its derivatives that make the applications easily maintainable by separating the application logic from the user interface. However, remote user interface protocols are not aware of the sophisticated capabilities of development frameworks that applications are using. As a consequence, the applications do not known that the user interface is rendered on a remote device.

Our framework changes this assumption allowing the developers of desktop and mobile applications to explicitly customize the appearance and behavior of the user interface rendered on remote devices, see Fig. 8.1. The application and the client become a network based system in which we leverage the MVC features of the application frameworks and the REST architectural style to have scalable and adaptive application sharing. With this approach we go beyond the classical

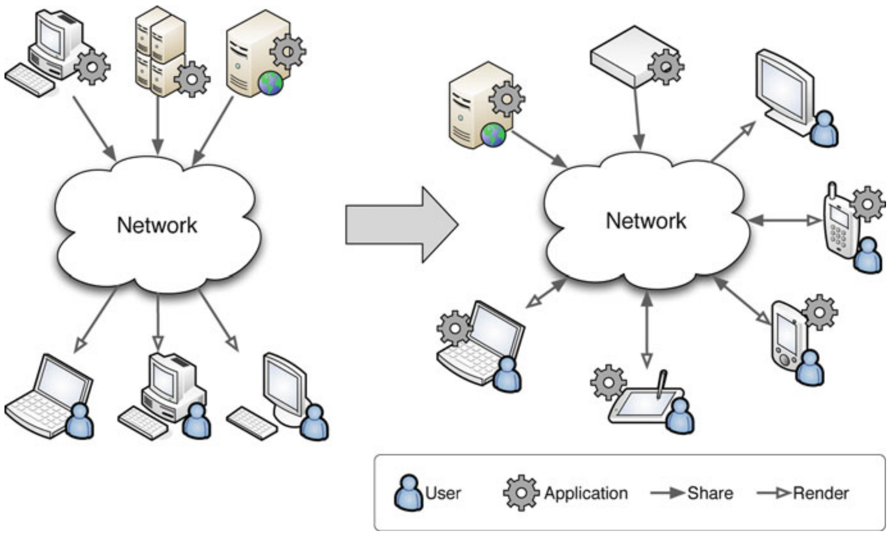

Fig. 8.1 Transition from traditional thin computing to an environment where applications can reside on any device and have the user interfaces rendered remotely according to the local look and feel

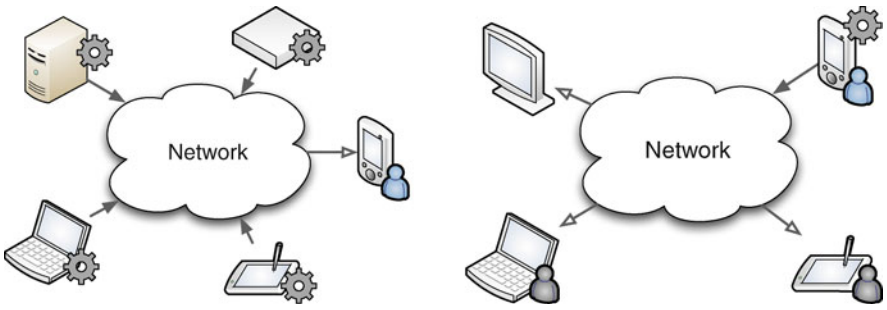

Fig. 8.2 Usage scenarios: application virtualization (*left*), and multi-display, multi-user applications (*right*)

application screen sharing scenarios in which the use interface is treated as a whole, enabling innovative applications that have the multiple user interfaces rendered remotely.

Usage Scenarios

The remote user interface paradigm can be applied in several ways depending on where the user and the application logic are physically located. From this perspective we can split the usage scenarios into two categories: *pull* and *push*. In the pull scenario, the user operates the device rendering the user interface, while in the push scenario the user operates the device where the application logic runs. These basic interaction primitives can be combined to tailor specific usage needs:

- **Application virtualization**. The application virtualization describes the situation in which the applications run on remote devices (e.g. personal computers, remote servers, consumer electronics or mobile devices) and the user interface is rendered on the device operated by the user. This scenario resembles closely the classical thin-client except that the exported user interface uses the local look and feel of the rendering device.
- **Multi-display applications**. The multi-display application scenario describes the situation in which the application runs on the device operated by the user and additional user interfaces are exported on nearby devices that render them using the local look and feel. Depending on the application context, these displays can act as secondary displays for the application or can be operated by additional users, a situation in which the application running on the remote device appears to the other users like a distributed cooperative application (Fig. 8.2).

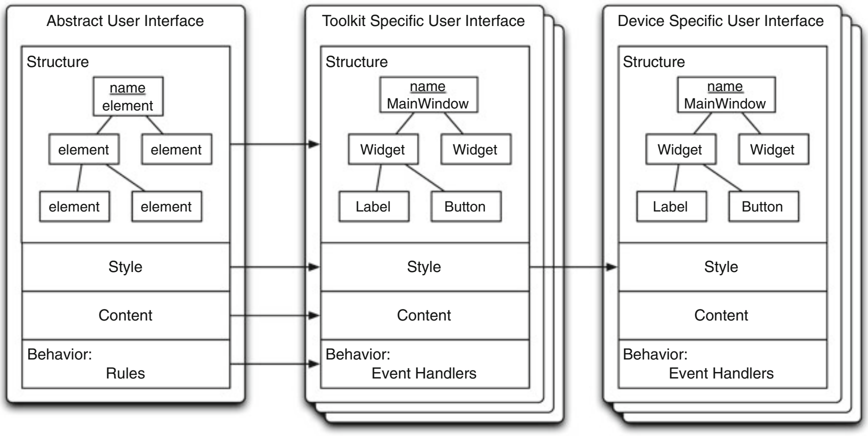

Fig. 8.3 Development process for scalable and adaptive user interfaces

Scalable and Adaptive User Interfaces

A scalable and adaptive interactive application is capable of exposing the user interface through multiple modalities and user interface toolkits, being able to adapt the user interface to the physical characteristics of the rendering devices. Developing adaptive and scalable applications is not trivial (see Fig. 8.3). To provide a good user experience the user interface needs to be designed with the particular device or device category in mind. Therefore, the user interface takes into account the features provided by the native user interface toolkit. A typical user interface can be divided into the following functional components: structure, style, content, and behavior. The structure contains the scene graph containing the elements of the user interface, with unique identification for each element. The style describes how the structure is presented to the users. The content describes which data is presented to the users in which elements of the user interface. The behavior describes what happens when the user interacts with specific elements of the user interface.

The scalable and adaptive user interface design process starts by creating the abstract user interface, which describes the user interface independently of any interaction modality or implementation technology. Its role is to capture the information that needs to be presented to the user, its structure and define the interaction behavior. Later, the abstract user interface is refined into toolkit specific user interfaces. They describe the user interface after a interaction modality has been selected (e.g. graphical). These user interfaces contain the final look and feel of the user interface by having the elements of the structure and the style mapped to the toolkit specific widgets, the content mapped to properties of the widgets, the layout of the widgets and behavior rules converted into event handler stubs. The user interfaces can be further refined for particular devices by adjusting the style parameters.

User interface description languages, such as UIML (Helms et al. 2008) and UsiXML (Limbourg et al. 2005), attempt to describe the user interface in declarative terms, without using low-level computer code. They aim at reducing the development effort by providing abstractions and automating the design process. Alternatively, SUPPLE (Gajos and Weld 2004) proposes a mechanism that automatically generates user interfaces according to device constraints. However, in practice, the challenges of having applications able to export the user interfaces to any device are significant (Want and Pering 2005). Therefore, we adopt a more relaxed approach that allows application developers to decide on what methodology to use for creating the user interfaces, and which devices they target.

From the User Interface to Web Resources

This section describes our approach for modeling the user interfaces of desktop and mobile applications as web resources. We start with an overview of the distributed system that enables applications to export their user interfaces to remote devices using the web architecture. Then, we describe in detail the functionality of the web resources that expose the user interface. Finally, we present the web-based mechanism that allows the remotely rendered user interface to be consistent with the state of the application, considering both the static dimension (e.g. user interface structure, consisting of elements and layouts), and the dynamic dimension (e.g. the values presented to the users at run-time) of the user interface.

Architecture Overview

The Model-View-Controller architectural pattern (MVC) separates the application engine that handles the data (e.g. the model) and the logic from the user interface that presents the model data, in a form suitable for the end user (e.g. the view), and handles the user input (e.g. the controller). The MVC pattern enables an interactive application to have multiple simultaneous views of the same model, allowing us to adapt the user interface to the various characteristics and form factors of the rendering devices.

The application sharing experience is provided by a system of two cooperating applications. The application to be shared provides the functionality of the model and the controller, while the user agent, which renders the user interface, provides the functionality of the view. In this distributed environment, the original interactive application becomes a network based system, which extends the classical MVC pattern with an event based mechanism that provides a level of consistency close to the case when the model, the view and the controller reside on the same physical device.

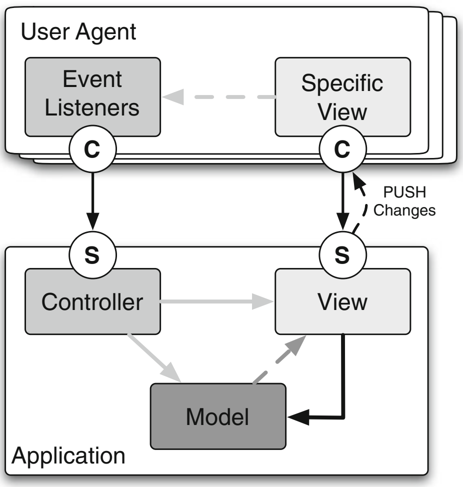

Fig. 8.4 Conceptual view of the Remote Model-View-Controller architecture

The Remote-MVC (depicted in Fig. 8.4) (Stirbu 2010) is based on a resource oriented architecture in which the user interface is exposed as a set of resources (e.g. view and controller resources). Each resource has a unique URI, is accessed using the methods of HTTP protocol and provides representations in well known formats such as XML specific to the rendering device or JavaScript Object Notation (JSON) (Crockford 2006).

The View-Related Resources

The User Interface

The user interface resource provides the first interaction point between the user agent and the application exporting the user interface (Fig. 8.5).

The `GET` method allows a user agent to acquire a user interface toolkit or device specific representation of the user interface. The user agent informs the user interface resource about its capabilities in the request using the `User-Agent` header and the HTTP content negotiation mechanism (e.g. the `Accept` header). The `User-Agent` identification string provides hints on the user interface toolkit version and on what device the user agent runs on, which can be further mapped to device form factor. The `Accept` header indicates the format in which the user agent accepts the representations. Based on the information provided by the user agent the resource implementation selects the most appropriate user interface, if any. The response body contains the toolkit or device specific representation of the user interface. Additionally, the header section contains links to the user interface element resources, to the event listener resources and to the monitor resource, which

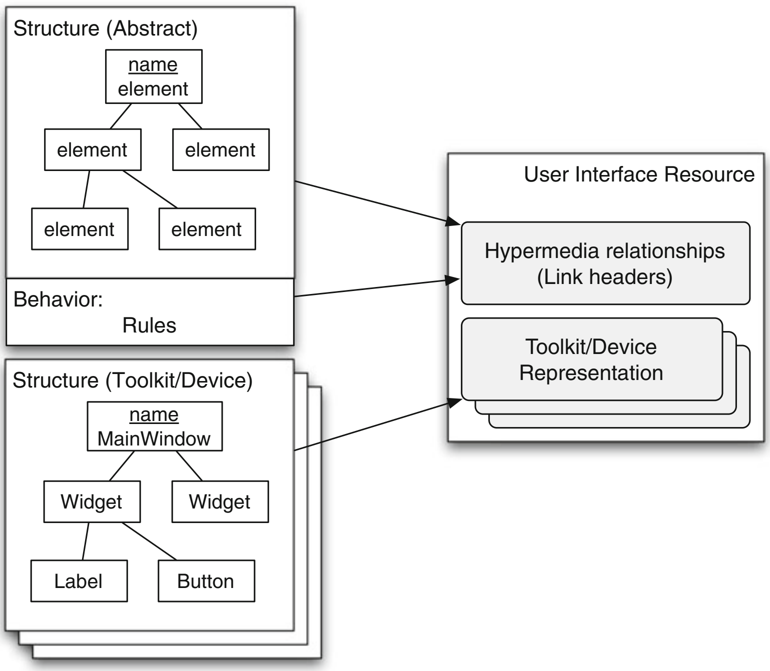

Fig. 8.5 Relations between the abstract user interface, toolkit/device specific user interfaces and the user interface resource

enables the user agent to be notified when the resource state changes. The URIs indicating the resources are provided using the `Link` (Nottingham 2010) header, and its relation `rel` to the current document are `widex.ui`, `widex.el`, and `monitor` (Roach 2010), respectively:

```
# Request
GET /appui HTTP/1.1
User-Agent: {User agent identification string}
Host: example.org
Accept: application/vnd.com.example.toolkit

# Response
HTTP/1.1 200 OK
Content-Type: application/vnd.com.example.toolkit
Link: </appui/uiHub/{uiElement}>; rel="widex.ui";
   uiElements="aUiElement,anotherUiElement,",
   </appui/elHub/{eventListener}>; rel="widex.el";
   eventListeners="anEventListener,anotherEventListener,",
   </monitor>; rel="monitor"

<!-- toolkit/device specific user interface representation -->
...
```

The User Interface Elements

A typical user interface is represented using a scene-graph data structure. Each node in this data structure represents an element of the user interface (e.g. a widget in graphical user interfaces), and each edge represents a parent–child relationship. Often, a node may have several children but only one parent.

The user interface of an application is determined by the structure of a scene-graph and by the properties of each node. We expose the application internal data structure that contains the view using a set of resources. By convention, we identify each resource corresponding to a user interface element using the following URI template:

```
http://example.org/appui/uiHub/{uiElement}
```

Although the URI scheme that identifies the user interface resources is flat, a typical toolkit specific representation of the user interface representation contains all information that allows a client to reconstruct the scene-graph structure. In case a toolkit specific representation does not have native support for describing the hierarchy of the user interface, we can overcome this limitation using use the XML linking language (Xlink) (DeRose et al. 2001) defined mechanism to encode the relationships between the resources.

The `GET` method allows a user agent to acquire the runtime values of relevant properties of the target user interface element. The properties of interest are indicated as the value of the `q` parameter of the query. If the query is missing, the server returns a list with all properties and their values that are considered relevant for the target user interface element:

```
# Request
GET /appui/uiHub/{uiElement}?q={paramName} HTTP/1.1
Host: example.org

# Response
HTTP/1.1 200 OK
Content-Type: application/json
Link: </monitor>; rel="monitor"
Link: </appui/uiHub/{uiElement}>; rel="edit"

{"paramName": value, }
```

If the value of a property has binary representation (e.g. an image), the response does not contain the value, but includes a JSON encoded link pointing to the content:

```
{
   "paramName":{
      "link":{
         "href":"/static/14411520525572505 6.png",
         "rel":"widex.static"
}}}
```

The POST method allows user agents to update the values of specific properties of the target user interface element. Typically, the argument of the call is a dictionary containing key-value pairs:

```
# Request
POST /appui/uiHub/{uiElement} HTTP/1.1
Host: example.org

{"paramName": value, }
```

The Controller-Related Resources

During typical usage, a user of an interactive application generates a large number of events. Among them only a small number are relevant for the application. Although all events emitted by the local window manager are passed to the application, the event handlers treat only the relevant ones, the rest being either ignored or handled by the widget implementations. For example, an application might be interested only when a button is pressed. This application has only one event handler (e.g. on_button_pressed) that handles the pressed event emitted by the button. The button widget implementation provided by the UI toolkit handles transparently the mouse movement through mouse move events and emits the pressed event only when the mouse is over the button and the mouse left button is pressed.

In our environment, it is not practical to transfer all events from the device rendering the user interface to the application host device, because the application handles there only a few of them. Each event handler defined in the controller is exposed as a correspondent event listener resource. The relevant listeners are provided to the user agent in the response to the user interface resource request. They are identified using the following URI template:

```
http://example.org/appui/elHub/{eventListener}
```

The PUT method allows a user agent to inform the controller that an event relevant for the target event listener was emitted on the user agent. The Controller is notified immediately when the message is received and the appropriate event handler is invoked with the provided parameters. Typically, the message body contains a list containing the values captured on the user agent by the listener:

```
# Request
PUT /appui/elHub/{eventListener} HTTP/1.1
Host: example.org

[aValue, anotherValue, ]
```

The Change Propagation Mechanism

Maintaining the state of the user interface synchronized with the application logic state is essential for an interactive application. In our network-based system, we use a change propagation mechanism to keep the user interface rendered by the user agent consistent with the application. The mechanism is driven on the server side by a special resource that notifies the user agent when representations change, and on the user agent side by the links embedded in responses from view and controller related resources. We first describe the monitor resource and then we describe how the change propagation effect is achieved in a way compliant with the REST architectural style.

The Monitor Resource

The view of an interactive application updates to reflect changes in the underlying model data. These changes are difficult to propagate to the user agents using only the request–response interaction pattern of the HTTP protocol, unless we use polling or long-polling techniques. However, these are expensive for the server who has to maintain open network connections for each user interface element resource. Instead we use a special resource that is able to stream notifications to the user agents whenever the representation of a user interface element changes, informing also how to obtain the change.

The `GET` allows the user agent to receive notifications whenever the user interface element resources change. The server streams these notifications over a long lived connection, borrowing characteristics from the `WATCH` method introduced in ARRESTED (Khare and Taylor 2004). Each notification is a link encoded in JSON representation (Allamaraju 2010). The user agent receiving such a notification performs a `GET` request on the target URI provided:

```
# Request
GET /appui/monitor HTTP/1.1
Host: example.org

# Response
HTTP/1.1 200 OK
Content-Type: application/json
Transfer-Encoding: chunked

{
   "link": {
      "href": "/appui/uiHub/aUiElement?q=aParamName",
      "rel": "widex.update"
}}
...
```

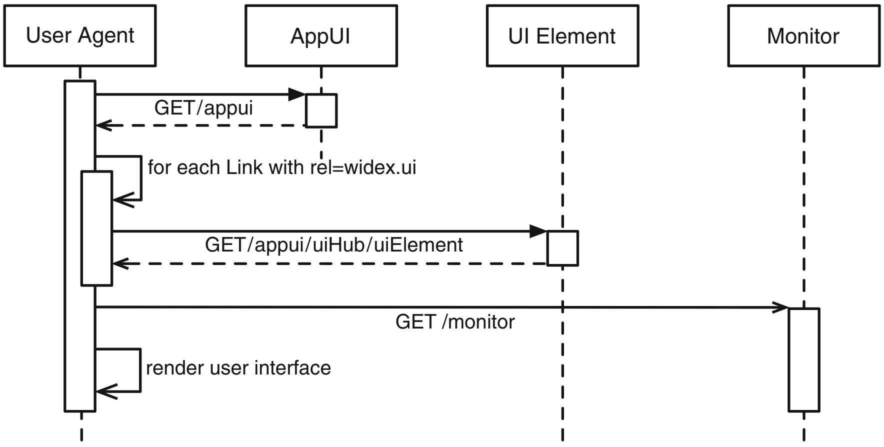

Fig. 8.6 Interaction pattern during the initialization of the change propagation mechanism

Orchestrating the Change-Propagation Mechanism

The change propagation effect is obtained as a result of cooperation between the user agent and the interactive application. The interactive application notifies the user agent when the content presented by the user interface elements change due to updates in the model, and the user agent notifies the application when the user edits the content presented by the user interface elements (e.g. edits content of fields in a form) or when the user interacts with specific user interface elements (e.g. clicks a button). The first phase of this process consists in the initialization of the user interface on the user agent side. The second phase involves the propagation of changes as they occur on the application side or on the user agent side.

The interaction pattern of the initialization is depicted in Fig. 8.6. At this stage, the user agent acquires a toolkit specific representation of the user interface, together with information that allows the user agent to acquire the current content presented by each user interface element, and which event listeners are relevant. The initialization ends when the user agent acquired the needed information and the user interface in its current form is displayed to the end user. The initialization process resembles a publish/subscribe scheme in which the application publishes the relevant resources and the user agent subscribes only to them.

The interaction pattern of the application triggered change propagation is depicted in Fig. 8.7. Whenever the content presented by a user interface element is updated by the model, the monitor resource notifies the user agent. The user agent acquires the new representation of the content by following the link provided in the notification and updates the rendered user interface with the newly acquired content.

The interaction pattern of the user agent triggered change propagation is depicted in Fig. 8.8. Whenever the user edits the content presented by a user interface element, the user agent propagates the change to the application by updating the

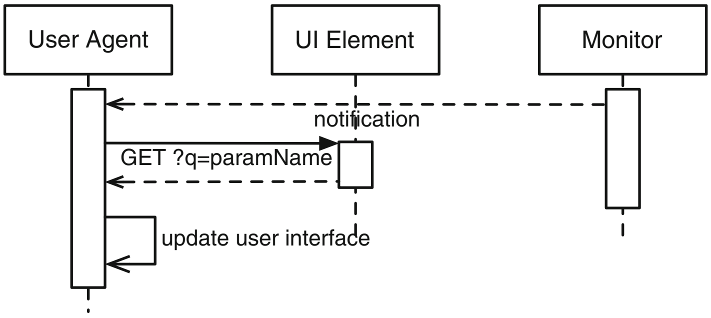

Fig. 8.7 Interaction pattern for application initiated change propagation

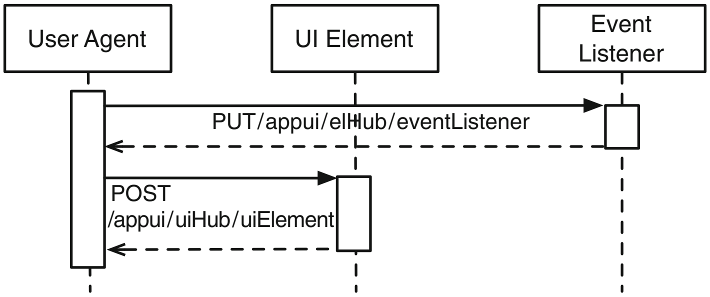

Fig. 8.8 Interaction pattern for user agent initiated change propagation

value on the corresponding resource exposed by the application. Similarly, the user agent notifies the application when a relevant event occurred by transferring to the application the event context characterized by the values corresponding to the event listener signature.

Prototype Implementation

This section describes the prototype implementation that enables scalable and adaptive sharing of Qt applications. Our implementation features the core components providing the server and the user agent functionality, and a set of add-on tools that speeds the task of creating and inspecting the network behavior of applications. The programming environment for application developers is Python, the bindings for Qt framework being provided by either PySide[1] or PyQt4.[2]

[1]http://www.pyside.org/.

[2]http://www.riverbankcomputing.co.uk/software/pyqt/intro.

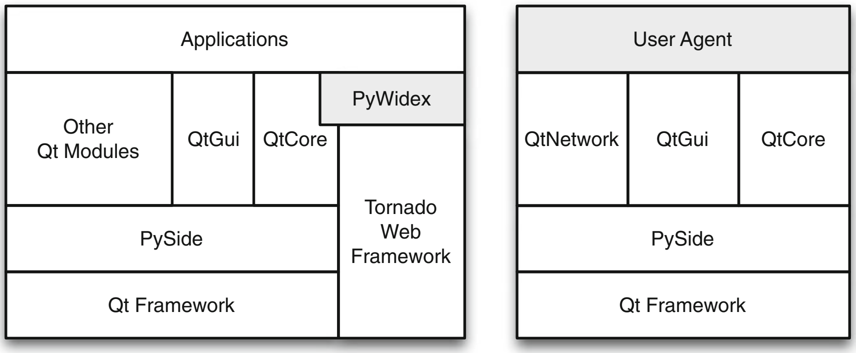

Fig. 8.9 The core components and their relation to the Qt software stack: application (*left*), and user agent (*right*)

The Core Components

The core components contain the basic functionality that enables scalable and adaptive sharing of Qt applications. The server functionality is provided by PyWidex (Widget Description Exchange). The package offers WebBackend, a convenience class that encapsulates all features needed to expose a view as a set of web resources, runs in its own thread, and is implemented as a web application running on top of Tornado,[3] a non-blocking, event-driven web server and RESTful framework optimized for real-time web services. Typically, an instance of this class is a property of the top level widget that provides the sharable view (Fig. 8.9).

The user agent is a standalone Qt application that is able to render the user interfaces exported by remote Qt applications, using the user interface toolkit specific to the rendering device. The user agent implementation is currently able to render user interfaces using multiple Qt user interface toolkits, e.g. QtGui or MeeGo Touch Framework. To render the user interface, the user agent application needs to know only the URI of the user interface resource of the view exported by the application. Once this information is known, the user agent configures itself using the information provided in the user interface representation.

Tools

Besides the core components, our prototype provides a set of add-ons that speeds the process of developing compatible applications:

[3]http://www.tornadoweb.org/.

- **Discovery**. The discovery functionality facilitates the creation and deployment of multi-display applications by enabling developers to specify on which kind of devices the user interfaces are to be rendered, and by allowing users to find instances of those devices in the proximity. A daemon running on the device allows remote applications to control the user agent. Currently the tool is based on zeroconf and support two user agent categories: public and handheld displays. Public displays automatically accept requests to render remote user interfaces while handheld displays are user devices, such as smartphones and tablets, which need user acceptance before a request to render the user interface is accepted.
- **Profiler**. The profiler tool provides tools for visualizing the interactions between the user agents and the applications.

Developing Multi-display and Multi-user Applications

In this section, we present the results of our experiments developing multi-display and multi-user applications using our middleware. We first describe how the test application works and then provide performance measurements.

Case-Study: Texas Hold'em Application

To demonstrate our middleware we implemented a Texas Hold'em application and used it in an environment containing consumer electronic devices (see Fig. 8.10) (Stirbu and Leppanen 2011). We have two users (e.g. Bob and Alice), each using a Nokia N900 smartphone, and a network enabled TV set having the back-end provided by a laptop running Ubuntu. In this setup, the TV set acts like a public display and Alice's N900 as a handheld display. All devices are connected to a local area network over Wi-Fi.

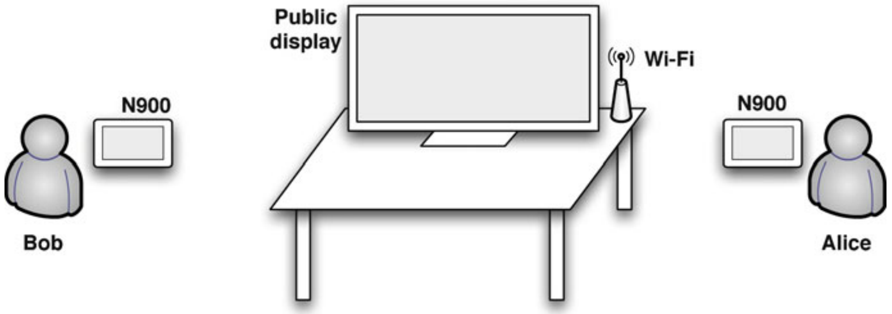

Fig. 8.10 Texas Hold'em: deployment environment

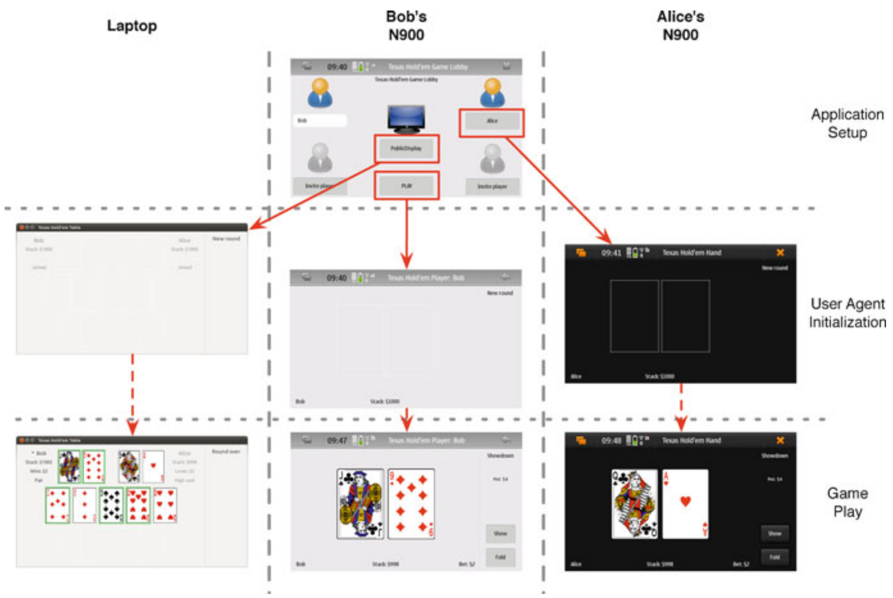

Fig. 8.11 Texas Hold'em: interaction flow

The Texas Hold'em is a multi-display and multi-user application. It has capabilities to accommodate up to four players, each presented with a view of their cards together with player specific information. Besides the player views, a public display presents the community cards and information of common interest about the game.

Figure 8.11 describes the typical usage scenario. Bob, the game host, starts the application on his handheld device. The application starts with the lobby view, which allows Bob to control where the views are displayed. First, he selects the public display, then invites Alice to join the game. The user agents on the public display and on Alice's handheld device connect to the corresponding table view and player view exposed on Bob's device. As the initialization is complete, the game can start. During the game, betting is controlled from the application engine by enabling and disabling the buttons in the appropriate player views.

Experimental Evaluation

The distributed interactive applications enabled by our system are presented to the user of each rendering device using the local look and feel. As each remote application looks like a native application, the users have similar expectations for the remote applications as for the local applications. In practical terms, the user interfaces of remote applications should have a reasonable startup time, and stay responsive in the intended usage environment.

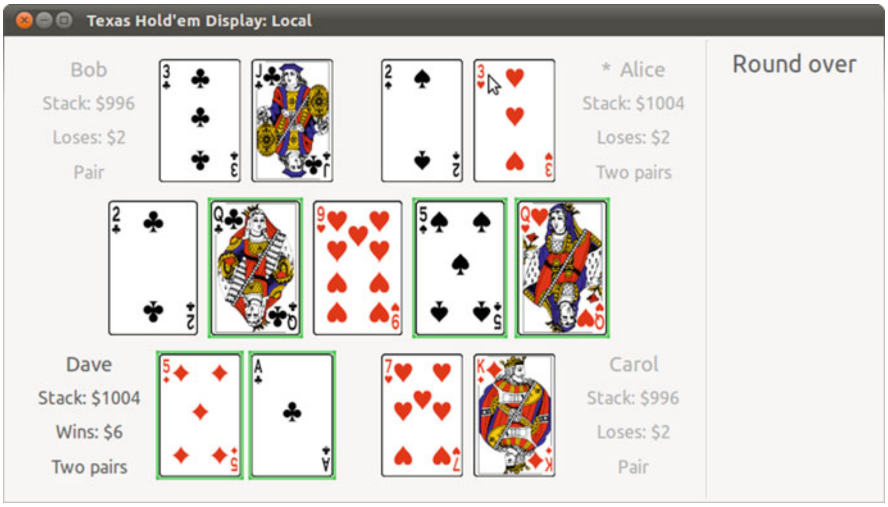

Fig. 8.12 User interface elements in the game table view

For any application, each user interface level interaction is typically implemented by the middleware using several network level interactions. For example, for initializing the user interface there is at least a request for each user interface element, which might add up to large numbers depending on how complex the user interface is. While using the application, each user interface change is implemented using one or two requests. Therefore, to evaluate the responsiveness of the user level interaction we need to aggregate the information from individual network interactions.

Among the views of the Texas Hold'em application, the game table is the most complex. The view presents information of common interest about the state of the game: four widgets for each user that plays the game, containing player specific information, one widget for showing the community cards and a side panel that provides information about the game phase. In total, the view contains 13 label widgets displaying images of the cards and 21 label widgets displaying text (Fig. 8.12).

The number of widgets that compose the game table makes this view a good candidate for determining how much time is needed by an user agent to render the initial user interface. Also, as the view contains common information about the state of the game, it gets updated frequently. Therefore, we use it for assessing how much time is needed to deliver the updates to the user agent.

We performed two round of tests. For the first test we used mobile devices for both Bob and Alice in an environment that simulates the intended usage scenario. For the second test we used laptops for all users with the intent of evaluating the impact of the hardware configurations.

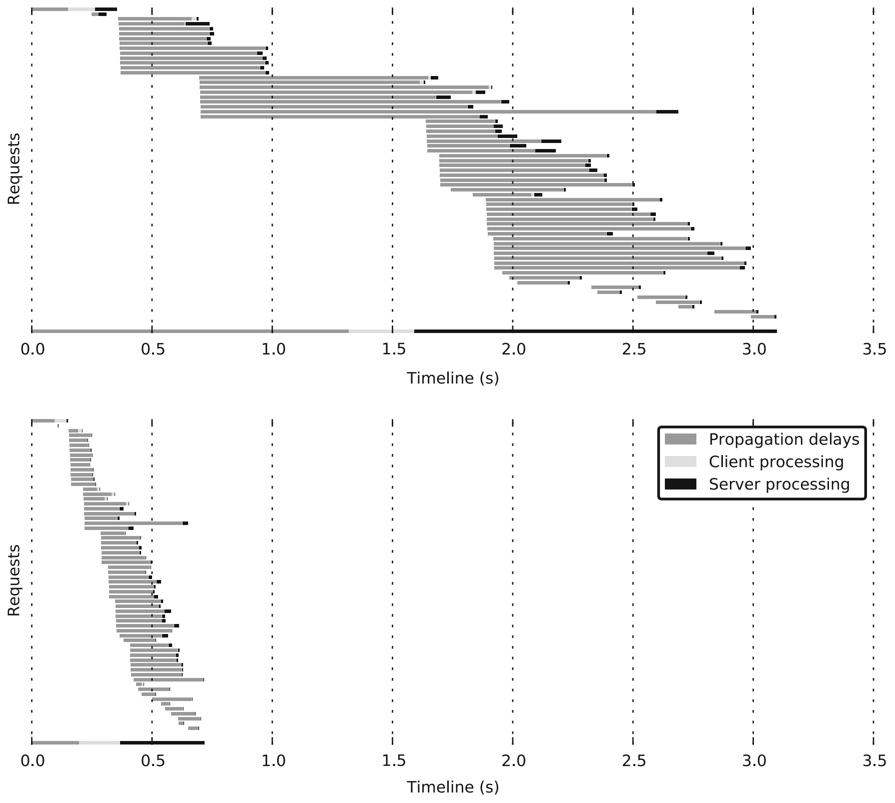

Fig. 8.13 Game table initialization: application running on N900 (*top*) and application running on laptop (*bottom*)

User Agent Initialization

While the user perceives the initialization of the user agent as one button click away, at network level the operation is decomposed into 67 requests. The interactions between the user agent and the application for initializing the game table user interface are described in Fig. 8.13. The top chart presents the interaction when the application runs on N900 while the bottom chart shows the interaction when the application runs on the laptop. Each interaction is broken down in three components each representing the time needed to process the request in the client, the server, as well as the delays induced by network propagation and the lower networking software stack.

The last bar in each chart represents the perceived initialization interaction. As both the server and the user agent are controlled by an event loop, operations associated with the requests are executed sequentially. This allows us to compute the total processing time for the client and server as the sum of individual request times. Due to HTTP pipelining, we are not able to determine the network induced

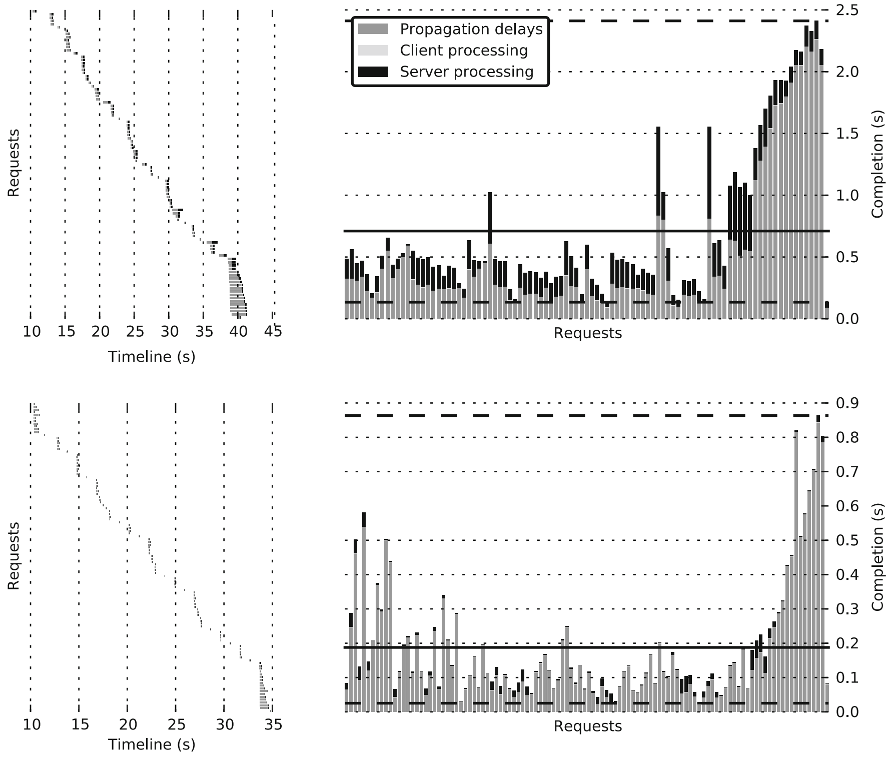

Fig. 8.14 Game table updates for a game round: application running on N900 (*top*), and application running on laptop (*bottom*)

delay, for completing the operation, directly from the information associated with individual requests. Therefore, we compute the perceived network latency as the difference between the time needed to complete the operation and the time used by server and user agent to process the requests.

User Interface Updates

We evaluated how fast the user interface updates are rendered on the user agent by having the game played by näive bots that advance to the next phase of the game after 2 s regardless of the cards in their hands. As for the initialization case, we first run the application on the N900 and then on the laptop. The user agents are the same in both test. The interaction pattern for each update is initiated by the application that notifies the user agent when a user interface element has changed, using the monitor resource. Then, the user agent request the new representation of the resource.

Figure 8.14 describes the interactions between the user agent and the application for both test scenarios as timeline and individual request completion time, each request being breakdown in time required by client and server to process the update, and network propagation delay. The beginning of the interaction represents the betting phase and is characterized by few updates, mostly related to one player at a time. The final part of the interaction corresponds to the showdown, a phase in which almost all user interface elements are updated in a very short time interval. This event resembles the user interface initialization, with the exception that the user interface structure is known and only the runtime values of the properties of the user interface elements are requested by the user agent, and the network propagation delay is longer due to notification delivery.

Discussion

REST is an architectural style. It contains a set of design guidelines, but it does not impose a specific architecture or a methodology. We follow these, but how RESTful is the resulting system? The Resource Oriented Architecture (ROA) (Richardson and Ruby 2007) introduces a practical approach for describing RESTful architectures through four concepts (e.g. resources, names as URIs, representations, and links between them), and four properties (e.g. addressability, statelessness, connectedness, and uniform interface). As the concepts are already covered, we'll focus the discussion on the required properties:

- *Addressability.* Our system exposes a resource for every piece of information that it serves about a sharable user interface: the structure using the user interface resource, the data presented to the user as user interface elements resources, and the behavior as event listener resources.
- *Statelessness.* Our goal of having the state of the rendered user interface synchronized with the application logic seems to go against statelessness. However, because the data of a sharable user interface is exposed at network level in a certain way by the system resources, the application web backend handles each HTTP request in isolation. All responses are generated based only on the information contained in the request. Having the networking substrate stateless, enables simpler generic implementation of the server functionality that can serve representations to user agents according to their needs.
- *Connectedness.* Resource representations in our system contain links to other resources. For example, the representation of the user interface resource has links to user interface elements and event listener resources, while user interface element resource may have links to static representations of binary content. These links point to subresources in the same realm, therefore the system is internally connected. Specific applications may include in their representations links to other applications or services.
- *Uniform interface.* Our resources can be accessed in a uniform way using the HTTP interface.

Additionally, the HTTP protocol provides established and widely supported mechanisms for negotiating content and caching. Content negotiation allows the user agents to acquire user interface representations that can be rendered using the local look and feel. Caching helps to reduce the network bandwidth requirements by not sending full responses when the user agent already has up to date representations.

Our system relies on the HTTP-based change propagation mechanism to synchronize the state of the user interface on the user agent and the application. To ensure that they share a consistent state, we rely on TCP to deliver the requests in an orderly fashion, and HTTP pipelining when TCP connections are reused. The result is a system that provides eventual consistency. For example, in the example Texas Hold'em application, the initialization and showdown may take several seconds to complete when the application runs on the handheld device but when the processing of the request burst is completed the state on both devices is consistent.

In general, the eventual consistency model achieved by our system is appropriate for interactive applications that can tolerate delays. However, it should be noted that the synchronization performance varies depending on the hardware capabilities of the devices used and the network distance between them. Application developers can use the profiling tool provided with the toolkit to identify bottlenecks and adjust the complexity of the user interfaces for the less capable devices, or to accommodate the expected network delays in the intended usage environment.

Security and privacy have not been addressed in this paper. However, we are investigating the use of oAuth (Hammer-Lahav 2010) to enable authorization of user agents and HTTPS for providing confidentiality protection for the communication between user agents and applications.

In this chapter, we presented our initial experiences on using REST for scalable and adaptive sharing of native desktop and mobile applications. Although our efforts are currently limited to Qt applications, the results are encouraging and we plan to enable using standard web browsers as user agents. However, these are just firsts steps towards our vision that applications can be experienced from any remote devices. More research and prototyping is needed before the vision is fully realized.

Acknowledgements We would like to thank Saku Tiainen for developing the Texas Hold'em application used for testing, and the referee who provided feedback and suggestions that improved the content of this manuscript. The research was completed in the Cloud Software Program[4] of the Strategic Centre for Science, Technology and Innovation in the Field of ICT (TIVIT),[5] and was partially funded by the Finnish Funding Agency for Technology and Innovation (TEKES).[6]

[4]http://www.cloudsoftwareprogram.org.

[5]http://www.tivit.fi.

[6]http://www.tekes.fi.

References

Allamaraju, S.: RESTful Web Services Cookbook, pp. 90–91. O'Rilley Media, Sebastopol, California (2010)

Crockford, D.: The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627, IETF (2006). Http://www.ietf.org/rfc/rfc4627.txt

DeRose, S., Orchard, D., Maler, E.: XML linking language (XLink) version 1.0. W3C recommendation, W3C (2001). Http://www.w3.org/TR/2001/REC-xlink-20010627/

Gajos, K., Weld, D.S.: Supple: automatically generating user interfaces. In: IUI '04: Proceedings of the 9th International Conference on Intelligent User Interfaces, pp. 93–100. ACM, New York, NY, USA (2004)

Hammer-Lahav, E.: The OAuth 1.0 Protocol. RFC 5849 (informational), IETF (2010). Http://www.ietf.org/rfc/rfc5849.txt

Helms, J., Schaefer, R., Luyten, K., Vanderdonckt, J., Vermeulen, J., Abrams, M.: User interface markup language (UIML) version 4.0. Committee draft, OASIS (2008). Http://www.oasis-open.org/committees/download.php/28457/uiml-4.0-cd01.pdf

Khare, R., Taylor, R.N.: Extending the representational state transfer (rest) architectural style for decentralized systems. In: ICSE '04: Proceedings of the 26th International Conference on Software Engineering, pp. 428–437. IEEE Computer Society, Washington, DC, USA (2004)

Krasner, G.E., Pope, S.T.: A cookbook for using the model-view controller user interface paradigm in smalltalk-80. J. Object Oriented Program. **1**(3), 26–49 (1988)

Limbourg, Q., Vanderdonckt, J., Michotte, B., Bouillon, L., Lpez-Jaquero, V.: Usixml: a language supporting multi-path development of user interfaces. In: R. Bastide, P. Palanque, J. Roth (eds.) Engineering Human Computer Interaction and Interactive Systems, *Lecture Notes in Computer Science*, vol. 3425, pp. 200–220. Springer, Berlin, Heidelberg, New York (2005)

Nottingham, M.: Web Linking. RFC 5988 (proposed standard), IETF (2010). Http://www.ietf.org/rfc/rfc5988.txt

Richardson, L., Ruby, S.: RESTful Web Services, pp. 79–105. O'Rilley Media, Sebastopol, California (2007)

Roach, A.: A SIP Event Package for Subscribing to Changes to an HTTP Resource. RFC 5989 (proposed standard), IETF (2010). Http://www.ietf.org/rfc/rfc5989.txt

Stirbu, V.: A restful architecture for adaptive and multi-device application sharing. In: WS-REST '10: Proceedings of the First International Workshop on RESTful Design, pp. 62–66. ACM, New York, NY, USA (2010)

Stirbu, V., Leppanen, T.: An open platform for distributed, scalable and adaptive interactive applications for CE devices. In: The 8th Annual IEEE Consumer Communications and Networking Conference – Demos (CCNC'2011 – Demos). Las Vegas, NV, USA (2011)

Want, R., Pering, T.: System challenges for ubiquitous & pervasive computing. In: Proceedings of the 27th International Conference on Software Engineering, ICSE '05, pp. 9–14. ACM, New York, NY, USA (2005)