A Revisit to Web Browsing on Wearable Devices

Jinwoo Song, Hyunjune Kim, Ming Jin and Honguk Woo
Software R&D Center, Samsung Electronics, Seoul, Korea, Republic of

Keywords: Wearable Devices, Fast Access Browsing, Widget View, Constrained Web Specification.

Abstract: Wearable devices and smartwatches have become prevalent in recent years, yet consuming web contents on those devices are not common mainly due to their restricted IO capabilities. In this paper, we revisit the web browser model and propose the notion of *fast access browsing* that incorporates the lightweight, always-on web snippets, namely *widget view*, into web applications. This allows smartwatch users to rapidly access web contents (i.e., within 200ms) similarly as they interact with notification. To do so, we analyse about 90 smartwatch applications, identify the quick preview pattern, and then define the constrained web specifications for smartwatches. Our implementation, the wearable device toolkit for *fast access browsing*, is now being tested and deployed on commercialized products and the developer tool for building *widget view* enabled web applications will be soon available as the smartwatch SDK extensions.

1 INTRODUCTION

Notification and application are two main smart experiences that wearable devices provide nowadays. The benefit of notification on wearable devices is considered well justified since the rapid access to contextually relevant information e.g., messages, appointments, and news, is usually important, yet running applications presumably need significant improvement. It is naturally anticipated that the smartphone users' experiences of playing with various applications and browsing web contents can be readily migrated to smartwatches and equally accepted, but unfortunately, our internal statistics on the app store activities indicates that smartwatch applications do not reach the wide popularity level of mobile app ecosystems.[1] Furthermore, in general, the existing platforms rarely prioritize the application of web browsing as an important feature on wearable devices.[2] It is mainly due to their inherent physical limitations such as tiny display and battery power that can degrade the browsing experience. For instance, Samsung Gear S2 (released on October 2015) smartwatch has 1.2inches 360x360 pixels display, 512MB RAM, and 300mAh battery while Samsung Galaxy S7 smartphone (released on March 2016) has several times capabilities including 5.1inches 1440x2560 pixels display, 4GB RAM, and 2550mAh battery.

Web browsing is a main part of daily life and gets more valuable as more time people are online through various devices and networks. It should be noted that web browsing is usually performed not only through conventional web browsers but also with web-enabled applications that are common in mobiles e.g., WebView applications of Android. In the aforementioned circumstance, however, web browsing would not be soon prevalent on wearable devices unless the way to dealing with web contents is essentially reformulated.

In this paper, we propose a new model for browsing web contents on wearable devices, specifically commonly available smartwatches, addressing the limitation of smartwatch applications and making the best use of the beneficial characteristics of notification, that is, the rapid access to the information. In doing so, (1) we first extend the web application structure by introducing a snippet of web contents, namely *widget view* that contains the most important contents provided by a web application and runs on the *always-on* mode for the fast access in a way analogous to how notification interacts with users. (2) We then analyse a set of existing smartwatch applications so as to

[1] Same insight can be found from *www.argusinsights.com/wearable-apps-2016*

[2] watchOS, Tizen wearable, and Android wear haven't included web browsers as *preload* applications on their commercial devices, and there is no smartwatch version of web browsers from major browser providers.

Song, J., Kim, H., Jin, M. and Woo, H.
A Revisit to Web Browsing on Wearable Devices.
DOI: 10.5220/0006232102120224
In *Proceedings of the 13th International Conference on Web Information Systems and Technologies (WEBIST 2017)*, pages 212-224
ISBN: 978-989-758-246-2

identify the information patterns for the fast access and define the web-based specifications for those patterns. Note that the specifications are not new but generally constrained from existing HTML/CSS/JavaScript. (3) We finally implement the concept of *fast access browsing* on smartwatches in which web applications can be provisioned and accessed through widget view.

As a result, loading the constrained web contents by widget views takes less than 200ms and has only 26 percent of memory footprint compared to when using a conventional web browser. In principle, such rapid loading and timely response satisfy the requirement of wearable specific swipe-based navigations similarly as notification does, while the constrained specifications do not much compromise whole web experiences. Our work has been tested on Tizen-based smartwatches. The runtime implementation for the constrained specifications is now being deployed commercially, and the developer tool for building *widget view*-enabled web applications will be soon available as the smartwatch SDK extensions.

2 BACKGROUND

Compared with smartphones' sophisticated applications, modern smartwatch applications have several limitations such as restricted touch interaction and runtime environment due to the fact that smartwatches have lower hardware capabilities and smaller displays (Apple, 2015, 'Apple watch human interface guidelines'; Samsung, 2014, 'Samsung gear application programming guide'; Connolly et al., 2014, 'Designing for wearables'). There have been several smartwatch OS platforms including watchOS, Android Wear and Tizen wearable profile, and they provide the common application types for similarly establishing the fast information access under such limitations: Glance of watchOS 2, Always-on app of Android Wear, and Widget app of Tizen. In the following, we analyse the application model of such three OS platforms, particularly concentrating on their common characteristics relevant to the fast information access so as to identify the concept and requirement of the fast access browsing.

2.1 Apple WatchOS

watchOS is the operating system of Apple Watch that provides the watch application project type consisting of two separately composed bundles, a Watch app and a WatchKit extension. A Watch app bundle contains the storyboards and resource files associated with the user interfaces of a watch application. A WatchKit extension bundle contains the extension delegate and the controls for managing those interfaces and responding to user interactions. Those bundles are packaged and deployed inside the iOS application on the mobile phone, and they are then installed on the user's watch and run locally as illustrated in Figure 1 (Apple, 2016, 'Apple watch app architecture').

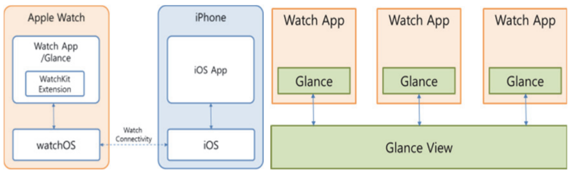


Figure 1: Apple Watch Application Structure.

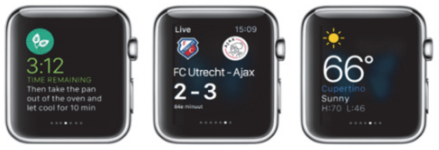


Figure 2: Apple Watch Glance Examples.

For supporting the fast access to important information, a Glance can be added in a WatchOS 2 project. Having a simple swipe at the bottom of the watch face, a user can quickly launch a Glance with a summarized view. Note that a Glance is part of a watch application and thus tapping a Glance usually makes its companion application displayed with the main interface in a full-fledged manner. The right diagram of Figure 1 describes the architectural view of Glance. Glances form a swipe-able collection of instant applications in that on each Glance, a user can quickly navigate to other Glances by swiping to left or right. Figure 2 depicts Glance examples that facilitate quick information view. (Bos, 2015)

Limitations

As implied by the name, Glances are meant to be quickly accessed and briefly looked, so there are several restrictions on how they can deal with contents. First, Glance contents and interfaces are intended for statically fitting on a single screen of a watch face. They are given non-scrolling in their UI structure and their text and graphical data items are set as read-only.

Moreover, Glances do not contain dynamic and interactive UI controllers and thus their functional capability is inherently limited. Glances do not intend for providing rich interactivity in that tapping

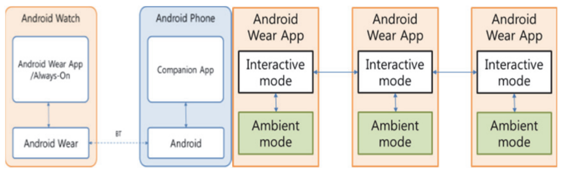


Figure 3: Android Wear Application Structure.

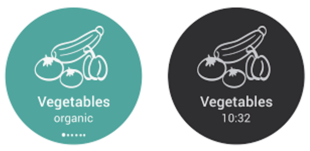


Figure 4: Android Wear always-on Application Example.

a Glance launches its companion application by default. Therefore only static contents are considered for Glance contents, indicating that buttons, switches, sliders, and menus are not supported. Furthermore, Glances are not able to directly access web contents. As the watchOS does not support the UIWebview controller that is commonly used for embedding web contents into iOS mobile applications. It is possible to have some workarounds to retrieve web contents, e.g., using a transcoding proxy that interacts with a web server and converts web contents to data streams that can be embedded in labels and image views. However, such a workaround requires additional implementation according to the specific rendering capability of watch applications and the result ends up with being not compatible with standard web architecture.

2.2 Android Wear

Android Wear is the Google's Android operating system specially tailored for smartwatches. An Android Wear application is packaged within a companion mobile application, and so a wearable application is automatically pushed and installed onto the Android Wear device while a user downloads and installs a mobile application from the Android store as illustrated in Figure 3 (Jeff, 2016).

Android Wear supports the low-power ambient mode by which the contents displayed on the watch face can be adaptively controlled for saving the battery power. In principle, an application can be configured as running in either such *ambient* mode for low-power operations or *interactive* mode (*normal* mode) with full functionalities. Note that the applications supporting both modes are categorized as *always-on*, and they are intended for keeping always visible; that is, even while a user drops her arm, an always-on application stays visible

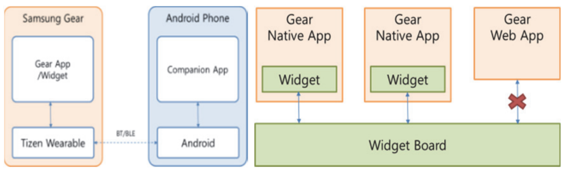


Figure 5: Tizen Wearable Application Structure.

on the watch face. The right diagram of Figure 3 depicts the always-on application structure and the navigation flow between applications. Figure 4 depicts an example always-on application, the shopping list in which the remaining shopping items are always shown even at the ambient mode.

Limitations

Similar to Glances, applications running on the ambient mode restrict their functionalities. First, the background color scheme is strictly limited to black, white, and gray. Second, the screen cannot be updated more frequently than every minute, so animations are not supported. For those applications that require more frequent updates, such as fitness, time-keeping, and travel information, developers may use *AlarmManager* object to wake up the processor and update the screen frequently but this is not recommended due to the overhead on the battery consumption. Moreover, navigation of an always-on application is restricted in that switching to other applications cannot be made by a single swipe on the ambient mode. An application needs to run on the normal mode before switching to another application as in Figure 3.

Current Android Wear does not support *WebView* component that is used for embedding web contents into Android mobile applications. This limitation is same as that of watchOS. There are downloadable web browsers for Android smartwatches (Google, 2016, 'Web browser for Android Wear') available from the Google Play but these independent browsers cannot be used for embedding web contents into other wearable applications.

2.3 Tizen Wearable

Tizen is the operating system based on the Linux kernel, being configured for supporting various device profiles including wearable devices. Samsung Gear devices based on Tizen wearable OS may be paired with Android mobile phones as in Figure 5. Tizen wearable applications can be written on both native and web APIs, and they can be packaged with .tpk (Tizen native package) and .wgt (Standard web application package) formats respectively.

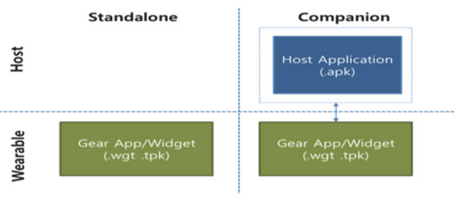


Figure 6: Tizen Wearable Application Structure.

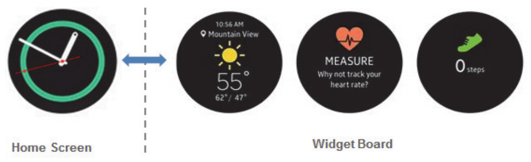


Figure 7: Tizen Widget Application Example.

A Tizen wearable application for Samsung Gear smartwatches can be configured to run on either the standalone mode or the companion mode (Samsung, 2015, 'Gear developer overview') as in Figure 6. Note that the standalone mode is not supported in either Android Wear or Apple watchOS. A standalone application runs independently while a companion application runs with two parts, a Gear application and a mobile host application (e.g., Android mobile application).

Similar to Glances of watchOS previously explained, Tizen supports a lightweight application model, namely Tizen widget that is used for the home screen customization and the quick approach to application functions. Widgets are displayed on the widget board as in the right diagram of Figure 5 therefore a user may navigate the widgets quickly from the widget board.

For a Gear device with the round design, widgets are located on the right side of the home screen and accessed by rotating the bezel. They offer important information and access to quick actins without requiring a user to open an application as shown in Figure 7.

Limitations

In Tizen, widgets are available for wearable devices. However, they are constrained in terms of the visible size, the types of interactions, and the maximum number of running instances. Generally a wearable widget takes the whole screen. Thus interaction events are restricted in that they are only available to distinguish widget events and platform events. Specifically rich interactions like vertical scrolling are not supported and tapping usually leads to open an application. Tizen allows up to 15 widgets on the widget board and each of them can be rapidly accessed from the right hand side of the home screen.

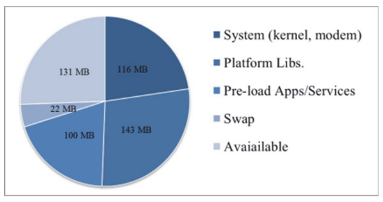


Figure 8: Memory Usages on Tizen Gear Device.

Different from the hybrid application model of Tizen by which both native and web APIs are used, widgets can be only written in Tizen native APIs as illustrated in Figure 5. It is the design decision of Tizen which considers the fact that multiple widgets should run as *always-on* and executing web APIs through a web engine (e.g., WebKit- or Chromium-based) for those widgets require much more memory than commercial products may provide. Our internal tests show that running 15 web-based widgets using the conventional WebKit-based web engine on a Tizen-based smartwatch product may consume 350~400MB runtime memory. In practice, this is hardly within the range of the available memory after the system boot-up when considering the 512MB RAM equipment. In fact, after the booting-up (running the system kernel, platform libraries and several preloaded native applications), there is only up to about 130MB available memory on those devices as in Figure 8.

2.4 Problem Definition

As explained previously, the modern OS platforms for smartwatches support the lightweight version of applications, watchOS 2 Glance, Android Wear Always-on, and Tizen widget, which commonly enable the fast access to important information and frequently used application features. These are effective, yet are limited of significance particularly in delivering web contents.

Our usability tests on the smartwatches significantly indicate that the rapid response for user input requests is critical in that e.g., switching the information among notifications and widgets should be done instantaneously with the swipe events on the watch screen. To enable such a rapid responsiveness, Tizen wearable generally maintains a set of notifications and widgets on the runtime memory, implying that runtime memory can easily be much consumed without a well-defined management policy. Suppose that a Tizen smartwatch has a runtime system where each widget consumes 5MB and it has 15 running widgets. In this case, out of 130MB available memory after the system booting-up, 75MB for the widgets are consumed.

It is our system safety configuration that 30MB is reserved for out-of-memory status. Considering these all memory consumption, the system ends up with 25MB available memory for running other applications. This calculation drives the requirement of memory usages when dealing with web contents on widgets.

In the following, under this consideration, we propose the fast access browsing system particularly dealing with web contents on smartwatches.

3 DESIGN OF FAST ACCESS BROWSING SYSTEM

Our proposed system for supporting the fast access browsing consists of three components: *mobile browser* that runs on the smartphones (which is not our focus in this paper), *widget view runtime* that manages the lifecycles of widget views, and *wearable browser* that supports the full-fledged browsing on non-constrained mode, e.g. with snap-scrolling (Rakow et al., 2016), of circular designs. The web browsing session of our proposed system is illustrated in Figure 9 where the interactions of three components are described below.

1. Suppose the mobile browser accesses a website *example.com* that has the widget view section in its web app manifest (Caceres et al., 2016).
2. The manifest is first sent to the wearable's widget view runtime and then gets installed.
3. The user chooses the installed widget view, adding it on the widget view screen. Note that the maximum number of widgets on the widget view screen can be configured depending on the device capability. The contents of the selected widget view (*example.com/widget_view.html*) are rendered and periodically refreshed according to the predefined configuration (e.g., once in 30min).
4. Upon a user event (e.g., touch interaction on the widget view), the widget view runtime launches the wearable browser with the pre-configured URL example.com for showing the richer contents.
5. The wearable browser loads the website from the URL and allows the wearable specific browsing experience.
6. Furthermore, the wearable browser can launch the mobile browser, if needed.

The *mobile browser* is capable of parsing the web app manifest associated with the website, and prompting a user for the widget view installation when it detects the wearable device being connected to the mobile device. Upon the user's consent, the manifest is sent over to the wearable device through an available connectivity such as Bluetooth. The *widget view runtime* manages the lifecycle of a widget view which has four states: installed, running,

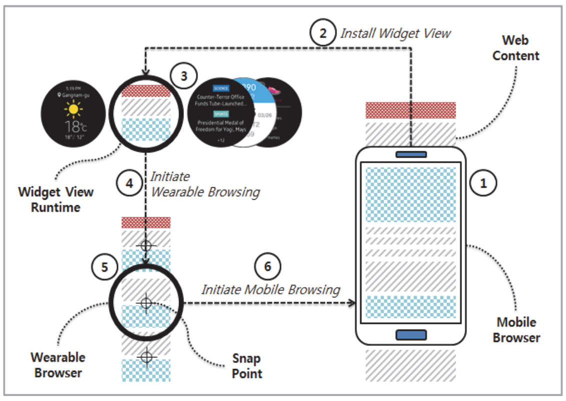


Figure 9: Overview of Fast Access Browsing System.

suspended, and uninstalled. After installation, the widget view appears in the widget view list. The user can select and put it in a position on the widget view screen.

As the widget view screen is configured to locate a set of widget views where the maximum number depends on the device capability such as the available memory for running always-on style applications. Thus, the user may need to remove one or more widget views from the widget view screen. The widget view runtime can be configured to periodically refresh the loaded widget view in order to fetch the latest information through the network. To save the battery power, the widget view is not be refreshed at the suspended state.

A widget view can be configured to trigger and launch the *wearable browser* application. Upon the trigger, the URL associated with the widget view is sent to the wearable browser over IPC, and the wearable browser loads the website for full-fledged browsing. Typically a well-designed website combines the CSS media query with its content styling so that the contents and layouts of the website can be optimized for small screen devices. One of the useful CSS styling techniques for small screen devices is the snap scrolling which allows the scrolling to stop at a predefined "snap point" position so that the user does not have to manually adjust the scroll stop position. When the wearable screen is too small for browsing, the wearable browser can send the URL back to the mobile browser. In this case, the is being sent over together with the URL so that the browsing experience on the mobile device can be seamlessly continued.

The overall system design enables the different modes of web content consumption: *full site* *browsing* experience with mobile browsing, quick glance experience with widget view, and *quick and richer* experience with wearable browsing.

Table 1: Application Domains.

	Applications
Health	Pedometer, Heartbeat, Drinking Water, Tracker
Planning	Scheduler, Alarm, Task Manger
Information	Weather, News, Stocks, Airline Ticketing, Traffic
Control	Music Play, App launch, System, Bluetooth

Table 2: Requirements for Widget View.

Layout Requirements	No-scrolling support Restricted interaction only with tap event Simple layout having text, font, or image Small screen with scrolling text
Feature Requirements	Sharing data between widget and application Timely data update via network connection Accessing and controlling system information Launching application

It is desirable that a user can choose her preferred way of web content consumption depending on the type of contents and contextual situations. Notice that the fast access browsing in this paper is introduced for web content consumption particularly on smartwatches but can be applied to other devices such as e.g., glasses with specification adaptations.

4 SPECIFICATION OF FAST-ACCESS BROWSING

4.1 Requirement Analysis

We analyse a set of representative smartwatch applications including 45 Tizen widgets, 35 watchOS Glances and 10 Android Wear Always-on applications whose domains are categorized as health, planning, information and control. The domains are denoted in Table 1. The applications in the health domain display the health-related data acquired from smartwatch sensors and update the data through a simple user interaction. The applications in the planning and the information domains generally display the application specific instant data, keeping them continuously updated through the network connection. Note that about the half of our analysed applications are in the information domain, e.g., weather, news, stocks, airline ticketing, traffic, etc. It is because smartwatches are considered particularly suitable for exposing the quick information to users and such benefits have been widely accepted in the market so far. Many applications in the information domain also provide the location-based information. There are also several control applications that play the music, control the system information, or launch the applications.

Table 3: Comparison of Wearable Features and Web APIs.

Feature	API	Health	Planning	Info.
DOM, Forms, Styles	HTMLF5 Forms	X	X	X
	Selectors API	O	O	O
	Media Queries	X	X	X
	CSS Transforms	X	X	X
	CSS Animations	X	X	X
	CSS Transitions	O	X	O
	CSS Color	O	O	O
	CSS Background/Border	O	O	O
	CSS Fexible Box Layout	X	X	X
	CSS Multi-col Layout	X	X	X
	CSS Text & Fonts	O	O	O
Device	Touch Event	X	X	X
	Device Orientation Event	X	X	X
Graphics	Canvas, SVG	X	X	X
Media	Video & Auido	X	X	X
	Web Speech API	X	X	X
Comm.	Web Socket API	X	X	X
	XMLHTTPRequest	X	O	O
	Web Messaging	X	X	X
	Geolocation	X	X	O
Storage	Web Storage	O	O	O
	Application Caches	X	X	X
	Indexed Database	X	X	X
Security	Cross-Origin Res Sharing	X	X	X
	iFrame	X	X	X
	Content Security Policy	X	X	X
UI	Clipboard API	X	X	X
	Drag and Drop	X	X	X
Perfor-mance	Web Workers	X	X	X
	Page Visibility	O	O	O
	requestAnimationFrame	O	O	O

Overall, these applications have the common feature that present information quickly to smartwatch users, and those are not web-based, but native on watchOS, Android Wear, or Tizen wearable. Based on this analysis, we take out the requirements regarding the widget layouts and the functional features so as to define the constrained web specifications for implementing widget views that render web contents. The requirements are summarized in Table 2. In the following sections, we explain the functional and the non-functional specifications that meet the requirements.

4.2 Functional Specification

Having the comparison of the application features and the W3C standard web APIs as shown in Table 3, we identify a subset of the web APIs that are necessary for achieving the fast access browsing through widget view. In addition, we include a subset of the device APIs that enable widget view to utilize the device native capabilities including

Application, File System, Sensor, System Information, and Message Port.

Application API provides a functionality to launch applications. File System API and Message Port API are used for the communication between widget views and application to share data. Sensor API provides the interfaces and methods for accessing internal device sensors. Sensor API is particularly used in the health domain. System Information API provides information about the device's display, network, storage and other capabilities that are useful in the control domain.

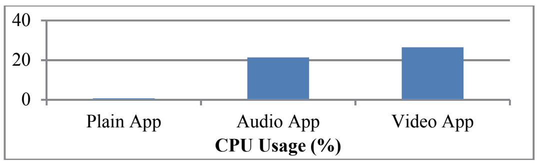


Figure 10: CPU Usage of Audio and Video.

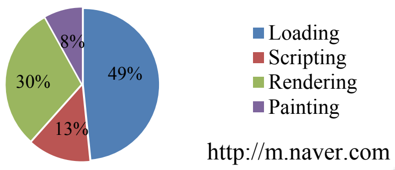


Figure 11: Loading Time Ratio for External Web Site.

Note that there are a couple of smartwatch applications that control the music playlist. However, we decide not to support the Audio and Video web APIs since they consume much CPU and battery resources. Maximizing the battery life is one of the most critical issues in the smartwatch usability (Min et al., 2015; Dredge, 2014; Proges, 2015; Rawassizadeh et al., 2014). Our experiments with different application types show that using the audio and video incur the overheads of more than 100 times compared to non-multimedia applications, as depicted in Figure 10. Therefore, we rather prefer exploiting mobile device resources to play the audio and video through the wearable interface.

4.3 Non-Functional Specification

In order to support the rapid loading of the web contents with small runtime memory and power consumption, we specify several non-functional restrictions and best practices including:

- Do not allow to load heavy resources such as CSS, JavaScript and images from external networks. In principle, loading those resources via networks not only takes much time with network roundtrips but it can frequently block from rendering web contents. Figure 11 illustrates the rendering pipeline time when browsing an example site (e.g., *m.naver.com*) and indicates that loading resources takes 49% of the complete processing time. This pattern would heavily change depending on the network connectivity but it is relevant since wearable devices may not be always in a stable network condition. AMP(Accelerated Mobile Page) recommends blocking all third-party JavaScripts so as to render web pages instantly (Google, 2016, 'How AMP Speeds Up Performance'). It is because third-party JavaScripts typically contains heavy processing codes. In the same sense, we restrict external resources when rendering widget views except for what is required to display the widget view contents. We consequently limit the specification that only XHR (XMLHTTPRequest) is allowed for directly interacting with external web resources, i.e., XHR is allowed to retrieve specific contents from backend web servers and update dynamically with the latest data.

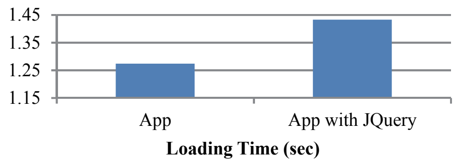


Figure 12: Loading Time by Different Resources.

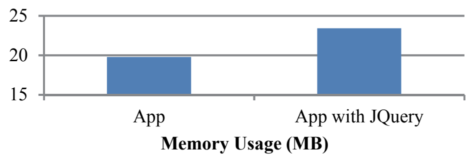


Figure 13: Memory Usage by Different Resources.

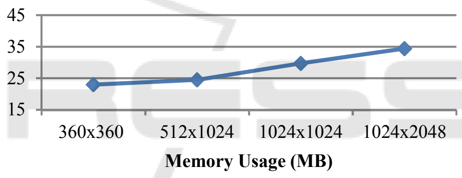


Figure 14: Memory Usage by Increasing Image Resolution.

Restrict the size of HTML, CSS, JavaScript files no more than 50KB. It is because our experiments illustrate that large resources, especially JavaScript, incur much loading delays as shown in Figure 12 and much memory consumptions as shown in Figure 13. We test several cases with different web frameworks including the jQuery library and find the same pattern in delays and memory usages. In the figure, the average application sizes are 23KB and

118KB respectively for base applications and applications with JQuery. The restriction is set as 50KB heuristically as it is generally sufficient to compose the UI layout and behavior logics of wearable widget views.

- Restrict the image resolution less than 1.5 times of the base image resolution. It is because higher resolution images consume much more memory as depicted in Figure 14. Also restrict the formats to the popular image formats such as JPEG, PNG and GIF. BMP is not allowed because the file size may be often too large.

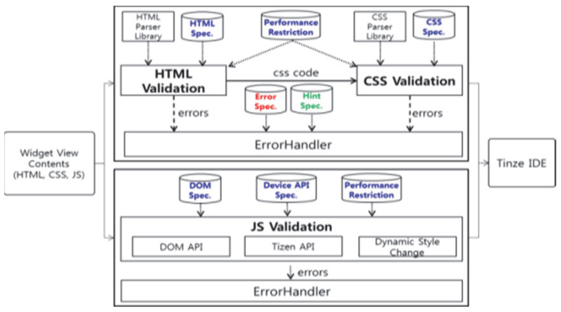
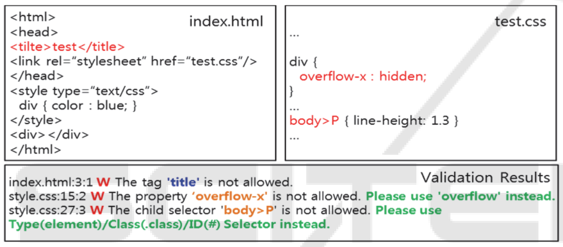


Figure 15: Widget View Validator.



```
<html>
<head>
<tilte>test</title>
<link rel="stylesheet" href="test.css"/>
</head>
<style type="text/css">
  div { color : blue; }
</style>
<div></div>
</html>
```

```
...

div {
  overflow-x : hidden;
}
...
body>P { line-height: 1.3 }
...
```

```
index.html:3:1 W The tag 'title' is not allowed.
style.css:15:2 W The property 'overflow-x' is not allowed. Please use 'overflow' instead.
style.css:27:3 W The child selector 'body>P' is not allowed. Please use
Type(element)/Class(.class)/ID(#) Selector instead.
```

Figure 16: Widget View Validation Example.

4.4 Specification Validation

As explained previously, the specifications of widget views are based on web standards but constrained in several ways. Therefore, if a code of widget views contains unsupported HTML, CSS or JS API, the widget view engine cannot interpret the code correctly. For helping developers in this situation, the widget view validator is additionally implemented as a pluggable function in the IDE.

Our implementation is based on Tizen IDE for wearable applications including widget views. The validator checks the code with the widget view specifications and then passes the results to the IDE as in Figure 15. Subsequently the IDE notifies the warning messages to a developer, if any. Note that the validator includes both the functional and the non-functional specification rules for checking.

Figure 16 illustrates a validation result of a sample widget view code with constrained HTML and CSS. The messages of the validator contain the file name, lines, columns, the warning messages, and the guide instructions so that developers can locate errors and use alternative APIs from the constrained specifications. In this example, CSS selector usages are guided to use 'Type/CSS selector' instead of restricted 'Child Selector'.

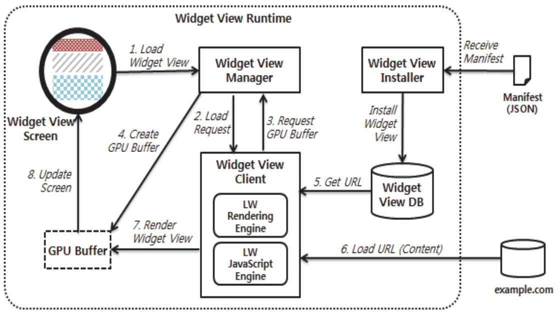


Figure 17: Widget View Runtime Architecture.

```
{
 "name": "My Daily News",
  ...,
 "start_url": "/index.html",
 "widget_view": {
      "widget_view_url": "/widget-view.html",
      "icon": {
         "src": "icon/news-widget-view.jpg",
         "size": "128x128",
         "type": "image/jpg"
      },
      "update_period": "60"
  }
}
```

Figure 18: Sample Widget View Manifest.

5 SYSTEM IMPLEMENTATION

In this section, we explain the system implementation of *widget view runtime* and *wearable browser extension* which are the major components of the fast access browsing system. Our implementation is based on Tizen wearable and deployed on Tizen-based smartwatches.

5.1 Widget View Runtime

The widget view runtime is the runtime environment by which widget views are installed, managed, and executed. Figure 17 depicts its internal architecture having three sub-modules: widget view installer, widget view manager, and widget view client, of which details are explained in the following.

Widget View Installer receives a manifest of a web application from the mobile browser and performs the subsequent installation steps. It is important that the associated URL of the manifest is delivered over a secure HTTP connection (i.e., HTTPS); this ensures for the fast access browsing system to retrieve the widget view from a trusted authority. The installation steps consist of parsing the manifest, extracting the relevant information such as widget view name, start-page URL, refresh period, etc., and registering the information into the entries database of widget views. It is worthwhile to note that the widget view installer is executed as a standalone process, as the process requires additional system privileges to perform a sequence of installation steps including validating the manifest, writing files to the system directory, registering to the package manager, and so on.

Figure 18 shows an example code of the widget view manifest that is slightly extended from the web app manifest (Caceres et al., 2016) to include the following data:

- "widget_view_url": refers to the content URL from which the widget view page is updated.
- "icon": specifies the icon of the widget view that is displayed on the widget view list menu.
- "update_period": specifies how often the widget view gets updated.

Widget View Manager is a daemon process that coordinates widget views and manages their lifecycle. The process is initialized at the system startup, establishing a communication channel with the widget view screen via the system IPC (inter-process communication). When a user selects a widget view from the installed widget view list, the widget view screen requests the widget view manager to load the selected widget view. The widget view manager in turn requests the widget view client to load and render the widget view on the GPU buffer which is shared between the widget view screen and the widget view client. As similarly to the widget view installer previously explained, the widget view manager has additional system privileges to access a set of sensitive system information including the installed widget list, the running widget list and their GPU buffer pointers, etc., and for this reason it runs on a separate process being isolated from widget view clients.

Each widget view follows the four states as in Figure 19. The widget view manager keeps track of the running and suspended states of the loaded widget views, and changes the states to be synchronized with the widget view screen (e.g., displayed on the widget view screen). The widget view manager also handles the update period of each loaded widget view. Indeed, due to the system-wide policy such that background processes are strictly limited in their CPU usage, updating a widget view in background (i.e., "suspended" state) is restricted.

If the update period expires and the widget view is in background (i.e., "suspended" state), this expiration is recorded by the widget view manager.

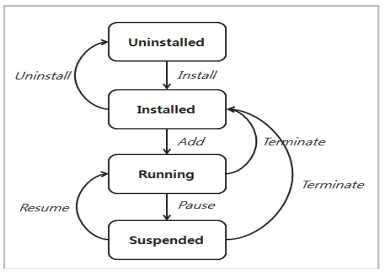


Figure 19: Widget View Runtime States and Transitions.

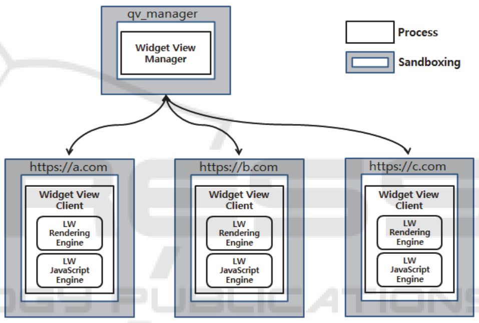


Figure 20: Widget View Client Process Sandboxing.

Later when the widget view transits from the suspended state to the running state, the widget view receives update event.

Widget View Client is a runtime process created by the widget view manager when loading a widget view is requested. To securely isolate the widget view execution and its possible crash among multiple executions of widget views and to reduce the system vulnerability, each widget view runs in a separate widget view client process. The process is sandboxed at system level so that it has strictly limited capabilities to access system calls, file system, and platform APIs. Our implementation exploits the security system policy of Tizen, SMACK (Simplified Mandatory Access Control Kernel) (Smack, 2011) for providing the fine-grained system level sandboxing. Figure 20 briefly shows the system architecture of the widget view client process.

SMACK implements the mandatory access control security as a Linux kernel security module. Executables and resource files installed on the system are assigned with corresponding security labels (also known as attributes). Whenever a process attempts to access a system resource, an authorization rule enforced by SMACK examines the security labels and decides whether the access can take place. For widgets, upon installation, the widget view client executable file (implemented as a soft link to the executable binary) and widget resource files are uniquely labeled by SMACK, and then a set of SMACK rules are generated and provisioned so that the widget resource files can be protected from illegal access from other widget contents running on separate client processes.

The lightweight rendering and JavaScript engine are implemented to render widget views in the constrained specifications explained in Section 4 and deployed as a shared library.

5.2 Wearable Browser Extension

The wearable browser works similarly as the mobile browser except for some wearable specific extensions which include CSS snap scrolling (Rakow et al., 2016) and CSS media queries. CSS snap scrolling allows web contents to be scrolled and stopped at specifically designed positions, called snap points. It aims at providing the better readability in a small display. CSS media queries are also extended for the circular shapes of wearable devices. Our proposed extension of CSS media queries is shown in Table 5. It is recommended that web contents are designed to be responsive to device capabilities. A typical smartwatch has less than 360pixel of viewport width which is different from that of mobile devices, and web contents can be adjusted to such a width through CSS media queries.

Furthermore, the wearable browser has the context menu (e.g., named as "Open in Mobile Browser") for launching the mobile browser. This menu is enabled only when a mobile device is connected. Clicking the menu sends the URL and the browsing session data such as the scrolling position to the mobile browser and lets a user continue browsing web contents across devices.

6 SYSTEM EVALUATION

We evaluated 32 widget view scenarios collected from both existing applications and developer requirements by implementing and running all the scenarios on Tizen-based smartwatches. The initial feedbacks from developers and test users are positive in that the specification meets the functional requirements. Regarding the non-functional requirements, we evaluate the memory usage and the loading time. We evaluate our implementation together with two different architectural configurations of a conventional rendering engine, WebKit. Figure 21 shows single and multi-process

Table 4: Media Query Extension.

@media (-geometric-shape: value)	
Value	rectangle or circle
Applied to	visual media types
Accept min/max prefixes	no

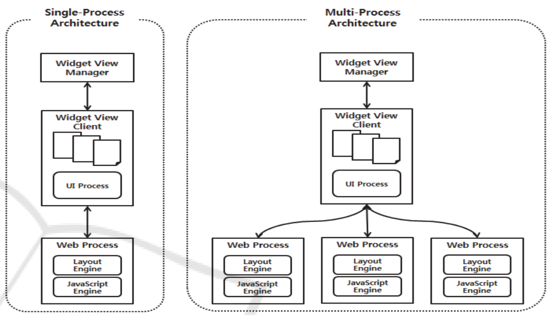


Figure 21: Single and Multi-Process Architecture Configurations with WebKit Rendering Engine.

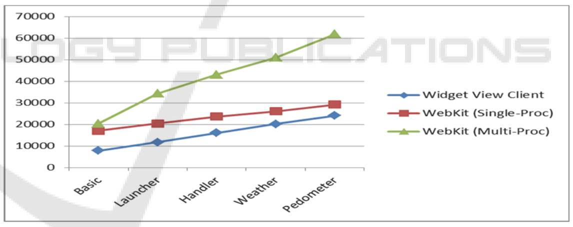


Figure 22: Accumulated Memory Usage of widget views.

architecture configurations of the widget view client with the WebKit rendering engine.

Figure 22 demonstrates the accumulated memory usage of loading five sample widget views on the widget view client, comparing with the single-process WebKit and the multi-process WebKit. In this experiment, each widget view is launched in sequence, from the basic to the pedometer, while previously executed ones are running. Widget views can run memory efficiently on the widget view client in that they require no more than 5MB on average for each execution whereas the single-process WebKit and the multi-process WebKit require about 6MB and 12MB respectively on average. Having this small memory consumption, the widget view runtime can keep running up to 20 widget views on a smartwatch within 512 MB.

It should be noted that while the single-process WebKit runs with less memory than the multi-process WebKit and slightly more than the web view client, it has been used only for comparison. Indeed, it is not preferred for commercial products. It is mainly because the single-process model does not provide the system level sandboxing, thereby not isolating the execution of a widget view from system

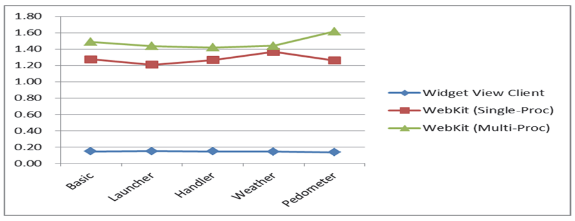


Figure 23: Widget View Loading Time.

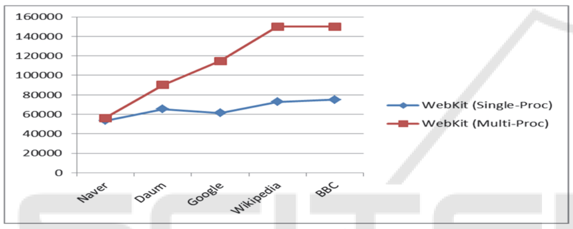


Figure 24: Accumulated Memory Usage of Mobile Websites.

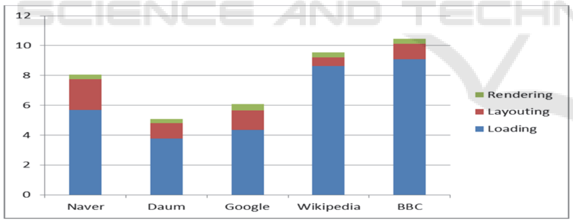


Figure 25: Loading Time Breakdown of Mobile Websites (Multi-Process WebKit).

crash and security vulnerabilities of other widget views. Chrome browser's multi-process architecture can effectively address the reliability and security problems that are common in real-world web contents (Barth et al., 2008; Reis and Gribble, 2009). Later WebKit community also adopted the similar multi-process architecture (WebKit, 2009). We include the single-process configuration in our comparison for evaluation purpose intentionally.

The loading time of widget views is shown in Figure 23. For loading sample web contents with constrained spec., the widget view client takes less than 200ms on average, while the single- and the multi-process WebKit take 1200~1500ms.

We test loading mobile websites with the single- and the multi-process WebKit configurations, and Figure 24 and 25 show the memory consumption and the loading time respectively. With the multi-process architecture, the system incurs the out of memory condition from the 4th website onwards, and in that case, loading takes up to 5~10sec to complete. From these results, we conclude that full-fledged mobile websites are often too heavy for wearable devices. The complicated web contents end up with the huge resource loading time and the relatively large memory usage in case of relying on a traditional web pipeline. The evaluation result implies that it is inappropriate to use a conventional rendering engine and support mobile websites by widget views.

7 RELATED WORK

Recently Google announced AMP project (Google, 2016, 'AMP Project') that allows developers to build web pages that are rendered instantly on mobile devices. AMP implements a set of custom HTML elements and a JS library that together bring the instant page loading performance. Some of optimization techniques adopted in AMP include disallowing of synchronous scripts, static resource sizing, CSS inline, style recalculation minimization, GPU-accelerated animations, etc. The custom AMP specification introduces a certain degree of learning curve to developers, and because of this barrier, AMP project provides a validation tool that allows developers to conveniently check syntax and performance errors. Despite the performance gain in loading time on mobile devices, both loading time and memory usage are not acceptable when it comes to wearable devices – the results shown in Figure 22 and 23 are not promising as the tested widget view pages are even lighter than typical AMP pages.

The Chromium project has a few on-going efforts to reduce Chromium Browser's memory usage (Chromium, 2015, 'Chromium Memory Team'). Some of these efforts include "compression of large string objects", "discarding layout trees", unifying internal allocators with a new allocator called "Oilpan", etc. It is not clear what will be the reduction rate of this effort when it comes to simple web pages like widget views, but we think it might not be that significant as for simple pages the heap memory allocated by the content will be much smaller than real-world websites and as a result the effect of memory reduction effort mentioned above might not be that significant.

Cobalt (2016) is a new lightweight rendering engine effort from Google that is compatible with a subset of the W3C HTML5 applications. It is built up from scratch an implementation of a simplified subset of HTML, CSS Box Model, and Web APIs that were really needed to build a full-screen, single-page web applications such as YouTube.com on constrained devices such as Smart TVs, Set-Top Boxes, Game Consoles, Blue-ray Disc Players, etc.

8 CONCLUSIONS

Smartwatches and wearable devices have gained much attention, yet there is no substantial improvement on delivering and rendering web contents on those devices mainly due to their restricted I/O capabilities. In this paper, we propose a new web browsing model with the constrained web specifications and the lightweight runtime based on the specifications which conjunctively provides the rapid access to web contents on wearable devices. The constrained web specifications are HTML, CSS, JavaScript with some restrictions that are based on our analysis on current smartwatch applications and focus on the fast information access.

The evaluation tests demonstrate that our work on recently commercialized smartwatches provides users with the well balanced experiences regarding functionality, expressiveness, and performance of web applications. Our future work includes developing a JavaScript framework and a server-based pub/sub broker system for providing a reliable performance of widget views with continuously updated contents. This work will be incorporating the concept of single page applications into the smartwatch runtime environments.

REFERENCES

Apple, 2015. *Apple watch human interface guidelines.* [Online] Available from: https://developer.apple.com/library/prerelease/ios/documentation/UserExperience/Conceptual/WatchHumanInterfaceGuidelines.

Samsung, 2014. *Samsung gear application programming guide*. [Online] Available from: http://img-developer.samsung.com/contents/cmm/Samsung_Gear_Application_Programming_Guide_1.0.pdf.

Connolly, E., Faaborg, A., Raffle, H., and Ryskamp, B., 2014. *Designing for wearables*. Google I/O.

Apple, 2016. *Apple watch app architecture*. [Online] Available from: https://developer.apple.com/library/ios/documentation/General/Conceptual/WatchKitProgrammingGuide/DesigningaWatchKitApp.html.

Jeff, 2016. *Architecture differences between wearable platforms*. [Online] Available from: http://blog.tizenappdev.com/2016/03/04/architecutral_differences_between_wearable_platforms/.

Google, 2016. *Web browser for Android Wear*. [Online] Available from: https://play.google.com/store/apps/details?id=com.appfour.wearbrowser.

Samsung, 2015. *Gear developer overview*. [Online] Available from: http://developer.samsung.com/gear/gear-develop-overview.

Caceres, M., Christiansen, K.R., Lamouri, M., and Kostiainen, A. , 2016. *W3C Web App Manifest Spec*. [Online] Available from: https://www.w3.org/TR/appmanifest/.

Min, C., Kang, S., Yoo, C., Cha, J., Choi, S., Oh, Y., and Song, J., 2015. Exploring current practices for battery use and management of smartwatches. In *Proceedings of the 2015 ACM International Symposium on Wearable Computers.* New York: ACM, pp. 11-18.

Dredge, S., 2014. *Apple Watch: battery life a challenge for a round-the-clock health tracker*. [Online] Available from: https://www.theguardian.com/technology/2014/sep/10/apple-watch-battery-life-health-tracker.

Proges, S., 2015. *These 4 Challenges Could Keep Smartwatches From Succeeding*. [Online] Available from: http://www.forbes.com/sites/sethporges/2015/02/25/these-are-the-4-challenges-keeping-smartwatches-from-succeeding/.

Rawassizadeh, R., Price, B. A., and Petre, M., 2014. Wearables: has the age of smartwatches finally arrived? *Communications of the ACM*, 58(1), pp. 45-47.

Rakow, M., Rossi, J., Atkins-Bittner, Tab., and Etemad, E.J., 2016. *W3C CSS Snap Scroll Spec*. [Online] Available from: https://drafts.csswg.org/css-scroll-snap/.

Barth, A., Jackson, C., Reis, C., and Google Chrome Team., 2008. The Security Architecture of the Chromium Browser. *Technical report*, Stanford University.

Reis, C. and Gribble, S. D., 2009. Isolating Web Programs in Modern Browser Architectures. *Proceedings of the 4th ACM European conference on Computer systems*, *Nuremburg,* New York: ACM, pp. 219-232.

WebKit, 2009. *WebKit2 High Level Document*. [Online] Available from: https://trac.webkit.org/wiki/WebKit2.

Google, 2016. *AMP Project*. [Online] Available from: https://www.ampproject.org/docs/get_started/about-amp.html.

Chromium, 2015. *Chromium Memory Team*. [Online] Available from: https://www.chromium.org/blink/memory-team.

Bos, R., 2015. *Designing for Apple Watch*. [Online] Available from: https://www.mangrove.com/en/journal/2015-02-25-designing-for-apple-watch.

Google, 2016. *How AMP Speeds Up Performance*. [Online] Available from: https://www.ampproject.org/docs/get_started/technical_overview.html.

Smack, 2011. The Smack Project. [Online] Available from: http://www.webcitation.org/6AqzohCXq.

Cobalt, 2016. The Cobalt Project. [Online] Available from: https://cobalt.googlesource.com/cobalt/.