Panelrama: A Framework for Cross-Device Applications

Jishuo Yang and Peter Hamilton
University of Toronto
Toronto, ON,
Canada, M5S 2E4
{jishuoyang, hamilton}@dgp.toronto.edu

ABSTRACT
We introduce Panelrama, a web framework for the construction of distributed user interfaces (DUI). Our implementation provides developers with low migration costs through built-in mechanisms for the synchronization of UI state and minimal changes to existing languages. Additionally, we describe a solution to categorize device characteristics and dynamically change UI allocation to best-fit devices. Finally, we illustrate the use of Panelrama through a sample application which demonstrates multi-device interaction techniques and collaborative potential.

Author Keywords
Distributed user interfaces, multi-device environments, optimization, user interface toolkit

ACM Classification Keywords
H.5.2. [Information Interfaces and Presentation]: User Interfaces - Graphical user interfaces

INTRODUCTION
Today, with the emergence of new computing form factors such as smartphones and tablets, it is increasingly likely that consumers own two or more computing devices. Currently, the majority of applications on these devices operate independently, with isolated application state and input and output (I/O) resources. The possibility of sharing state information and I/O resources between devices opens up a wealth of interaction possibilities: a user may retain workflow between devices, use additional hardware to overcome device limitations, or use additional devices to enrich interactions. Existing works make use of distributed user interfaces (DUIs) to allow applications to take advantage of multiple devices. DUIs enable developers to distribute portions of the application interface to a number of devices, users, displays, and platforms [11].

In order to minimize the migration effort in making existing applications multi-device friendly, we developed our framework Panelrama with the goal of implementing DUI by making modest modifications to existing development tools rather than formalizing new standards. We observed that UI elements are commonly arranged in logical groups that should remain together (e.g. media controls for a player). We provide a new XML element, which we have dubbed panels, which may be placed around such controls.

Once decomposed as a set of panels, an application can then be further modified to specify state information which should be synchronized across devices (e.g. position of the seek bar is synchronized to the progress of the video). Panelrama automates synchronization of state information, as well as the ability for developers to customize how ad hoc synchronizations should be handled.

Panelrama recognizes that the constituent elements of a given panels may make it more or less suitable for a given device (for example, the video window is more suitable to a television than a phone). Thus, in addition to allowing the splitting of an interface, our framework allows developers to specify the suitability of panels to different types of devices. This allows our optimization algorithm to distribute panels to devices which maximize their usability. In the following sections, we explain the Panelrama architecture, its automatic allocation algorithm, and our sample application.

RELATED WORK
The use of distributed interfaces spanning multiple devices is not a new idea. Like Panelrama, previous works such as DIAMOND [8] and XICE [1] provide frameworks to help developers split the interface for a single application across different devices, but these frameworks involve heavy use of native code, making them difficult to deploy across platforms. Other existing frameworks such as WebSplitter [7] and various works from Paternò et al [4, 5] take advantage of the web browser to provide cross-platform support. These commonly use XML tags to divide an interface at a very fine-grained level, and gives developers a high degree of control over UI distribution [10]. However it also increases the cost of migration, requiring considerable rework for larger applications to use these frameworks.

Characteristics of certain UI elements make them more suitable to some devices rather than others, therefore a distributed UI framework must take care to optimize the distribution of UI. Existing works such as SUPPLE [3] and other adaptive interfaces [2] focus on the retargeting of UI through the generation of UI elements suited to the device on which they are located. Often the new interface is produced through a constrained optimization algorithm. These approaches either abstract the UI adaptivity away from the developer [3], or expose the developer to new markup languages which make migration of existing applications difficult [2]. We have not focused on UI retargeting. However, we draw inspiration from these works in developing our automatic panel distribution mechanisms.
In addition, the adoption of distributed UI also requires the backend architecture to support the state information of UI elements across devices. This is not addressed in existing web frameworks like WebSocket [7], since it can often be assumed that the developer is responsible for the backend implementation. We have found, however, that by building state synchronization and transfer into the framework itself, it is possible to reduce migration costs and code changes.

From these existing approaches, we determine that while current distributed UI frameworks are powerful and allow for a degree of control, at the same time they necessitate sweeping changes to existing applications. We identify a need for a framework to provide the benefit of distributed UI across devices, while simultaneously being simple to learn and adaptable to reduce the cost of migration.

ARCHITECTURE & DEVELOPMENT EXPERIENCE
Panelrama provides users and developers with the ability to allocate and transfer panels, as well as synchronize shared panel states across multiple devices. A key guiding principle was to avoid significant changes to traditional application development, ensuring that porting of existing applications would be straightforward. To use Panelrama, a developer needs only to:

1. Wrap groupings of UI elements into a new 'panel' object
2. Select state information for synchronization
3. Write ad hoc synchronization 'merge' function (if needed)
4. Rate each panel for how well it takes advantage of various device characteristics (for automatic panel distribution)

Framework Dependencies
Panelrama is built atop the Meteor framework [9], a JavaScript framework designed for easy development of reactive web applications using dynamic HTML and NoSQL databases. Meteor combines three technologies: Node.js to automate packaging and distribution of code assets, NoSQL database MongoDB to simplify and speed up data storage and access, and Handlebars templating framework to provide dynamic HTML containers. Unlike many existing application frameworks, Meteor utilizes imperative programming, so that results to code are "automatically recalculated whenever data changes" [9].

We utilize Meteor's HTML templating and reactivity to propagate state changes across panels. The user-facing side of a Panelrama panel is represented by a Meteor template, whose contents are updated reactively to changes in its underlying state information stored in Meteor's MongoDB database. Device states are similarly monitored: when a new panel is assigned to a device, Meteor dynamically renders the panel's HTML on the device in question.

Panel Creation & Shared State Information
The code assets for defining a Panelrama panel are divided into three categories: HTML/CSS, JavaScript functions, and a Panelrama-specific panel definition. This design makes it easy for developers to port existing web applications to take advantage of Panelrama, since pre-existing assets in HTML/CSS and JavaScript are reusable, and only require minor changes to support Meteor's templating framework.

To define a new type of panel, the developer wraps each portion of UI HTML to be grouped into a panel with Meteor's template tags. Since the id and class properties of the HTML elements are unaltered, developers can reuse existing JavaScript functions that refer to the HTML elements. After dividing the HTML into templates, the developer would next create a short Panelrama-specific panel definition by extending Panelrama's default Panel object. Extending the default Panel object gives the new panel type access to Panelrama's built-in panel allocation functions, these include: addition, removal, transfer, and cloning of panels. A complete definition includes: naming the new panel type, default state information for new panels of this type, and this type of panel's allocation weight for different device characteristics (e.g.: Figure 1).

```javascript
function MapPanel(id) {
    this.type = "MapPanel";

    this.stateVariables = {
        longitude: {value: 0, sync: true},
        latitude: {value: 0, sync: true},
        zoomLevel: {value: 1, sync: true},
        layer: {value: "roadMap", sync: true}
    };

    this.allocationWeight = {physicalSize: 8, keyboardCapability: 0, touchCapability: 3};

    Panel.call(this, id);
}
```

Figure 1. A panel definition of a satellite map panel.

The panel's state information represents the portions of the application's model mapped to the panel's view. This enables Panelrama to update the HTML view of panels to reflect any change in the state information of the overarching cross-device application model. Each state variable contains both its value and a boolean flag indicating whether the variable is currently being synchronized with the global state. This is to provide a built-in solution for synchronizing panel states between devices, and will be explained in detail in the next section.

Panel Synchronization
A key to enabling applications to easily span devices is to provide a mechanism to distribute the global state of the application. Panelrama provides developers with a built-in solution for synchronizing panels through a shared global state, synchronized via the state variables seen in Figure 1.

As shown in Figure 2, a Panelrama session includes a list of all connected devices and a list of all available panels. Each device is assigned a certain number of panels, we refer to these panels as local panels. Additionally, for every type of panel, we create a global panel that is not assigned to any device. The developer is given the freedom to choose which state variables a given local panel should sync with the...
global version using the “sync” flag (see Figure 1). When a panel’s state variable is synchronized, any updates to that state automatically change the associated global state (and vice versa), while an unsynchronized state variable does not propagate its changes to the global panel, nor does it receive updates when the global panel is changed. For example, in Figure 2, the current video and position are sync’d on the desktop computer’s local panel (indicated with a green arrow), while the others are not (red arrows).

![Figure 2. Architectural overview of Panelrama](image)

Developers are also free to only occasionally sync certain variables. To support this, Panelrama provides three methods:
- **Push** (overwrite global state with local state)
- **Pull** (overwrite local state with global state)
- **Merge** (a developer-specified resolution)

For example, in a video streaming application, such as the one shown in Figure 2, a synchronized video panel may be shown on the largest screen (e.g.: a TV) for group viewing, but use private unsynchronized video panels for previewing videos on tablets or phones. The developer can provide a ‘show on TV’ button which initiates a push synchronization to the global state, which would immediately cause the TV to show the video seen on the tablet. The developer could also elect to sync the ‘location’ field of the tablet and TV videos, allowing the tablet to act as a remote control.

The behavior of the merge synchronization is based on application logic. For example, in a multi-user calendar application, a developer might use merge to produce a shared calendar showing common available times for all users. This is done by overriding merge in the base object.

**AUTOMATIC PANEL DISTRIBUTION**

In addition to synchronizing states between panels, Panelrama enables developers to automatically distribute panels to best-fit devices. In order to take advantage of our optimizer, we rely on two key pieces of information. First, various device characteristics, such as screen size or input methods, are required. Second, a representation from the developer of the importance of each characteristic to the usability of a given panel. For example, in a video player, screen size might be matter less for a playback control panel than for a video playback panel.

Similar to the approach used in SUPPLE [3] to adapt single-device interfaces, we cast our distribution algorithm as a linear optimization problem, intended to match the needs of panels to the capabilities of available devices.

**Device and Panel Modeling**

We model a device $D$ and a panel $P$ as two tuples:

$$D \equiv (S, C_d), P \equiv (C_p)$$

where $S$ is panel capacity of the device, and $C_d$ represents the set of device-specific characteristics (e.g. resolution, physical size, input method). A device may not be allocated panels exceeding the number specified by $S$. On a panel, $C_p$ is a set identical in size to $C_d$, and represents the set of weights of device requirements with respect to the panel.

In Panelrama, both $C_d$ and $C_p$ are represented as a set of integer scores ranked on a standardized scale (such as from 1-5). For any given application, Panelrama provides the ability to determine a subset of $C_d$ programmatically, such as assigning a score to hardware resolution. The remaining $C_d$ scores, as well as all $C_p$ scores, are determined by the developer. For instance, a panel requiring a large device may be assigned a $C_p$ score of 5 with respect to physical size. Panelrama provides a list of sample device categories with preconfigured $C_d$ scores. Developers may choose to add additional device categories, support additional device characteristics, or override any of the existing $C_d$ scores.

To calculate the cost function for the linear optimization problem, we treat each $C_j$ and $C_p$ as a vector. For every combination of $C_d$ and $C_p$, we calculate the dot product of the vectors and them in a cost matrix. Given $P$ panels assigned to $D$ devices, we get a cost matrix $C$ of size $D \times P$, where $C_{dp}$ is the usability score of panel $p$ allocated to device $d$. These scores form the basis of the objective function used to solve our optimization problem.

**Linear Optimization Function**

Formally, we represent the optimization problem as a linear sum assignment problem: there are $P$ panels assigned to $D$ devices. Given cost matrix $C$, the usability score of panel $p$ on device $d$ is $C_{dp}$ for $d = 1, ..., D$ and $p = 1, ..., P$.

We will want to arrive at assignment $x_{dp}$ which maximizes the usability score, for $d = 1, ..., D$ and $p = 1, ..., P$, where $x_{dp}$ represents the allocation of panel $p$ on device $d$. Therefore our objective function is:

$$\text{maximize } \sum_{d=1}^{D} \sum_{p=1}^{P} C_{dp} x_{dp}$$


Finally, no panel $p$ may be assigned to more than one device $d$. No device $d$ may be allocated panels exceeding device capacity $S_d$. Therefore we have the constraints:

$$\sum_{d=1}^{P} x_{dp} \leq 1 \text{ for } p = 1, \ldots, P$$

$$\sum_{p=1}^{D} x_{dp} \leq S_d \text{ for } d = 1, \ldots, D$$

Whenever a panel or device is added or removed, Panelrama encodes this optimization problem into CPLEX LP format and computes the answer using a JavaScript version of the GNU Linear Programming Kit [6].

**EVALUATION**

**Synchronization Performance**

For a web-based application, it is important for the state information to be propagated quickly. However, given the number of devices a user may choose to connect to a Panelrama session, we need to determine how well Panelrama synchronization scales with additional devices. We ran our test with a video playback application deployed on a remote web server hosted in Ashburn, VN, USA, and issued play and pause commands on a single device located in Toronto, Canada. We connected a varying number of devices to the Panelrama session and measured the average time required for these devices to have their state changed after issuing the command. As seen in our results from Figure 3, the synchronization time remained constant as number of connected devices increases.

**Optimization Performance**

Ideally, transitions involving addition or removal of panels should be seamless to the user. Since the optimization function is called whenever the configuration of devices or panels is changed, the algorithm needs to execute quickly. We conducted performance tests of the solver, varying numbers of panels and devices and measuring the time required for calculation. We kept one variable constant at 8, a representative number for the majority of our single user applications, while increasing the other variable. The tests were run using a laptop computer with a quad-core Intel i7 at 2.0GHz and 8GB of RAM. The results are seen in Figure 4. We see that with our modest ‘server’, 8 devices can run up to 40 panels, and 8 panels can be distributed across up to 40 devices, in less than 50ms. Given the latency typical of web applications, we consider this value to be acceptable.

**VIDEO STREAMING APPLICATION**

In order to better demonstrate the capabilities of Panelrama, we developed an application using Panelrama and the Google YouTube API. Our shared YouTube browser includes four panels: a video stream panel, a playback controls panel, a search panel, and a related videos panel.

The video panel holds state information such as the current playtime, the playback status (playing, paused), and the video's YouTube identifier. These states are controlled by the three remaining panels. Panelrama's synchronization mechanism allows the distribution of a synchronized video panel to a television, and multiple users to interact with the global video panel using controls on phones or tablets. The user also has an option to add a personal video panel that is not synced to the group video, allowing the user to search for and preview videos without interrupting the group. When the user has selected an appropriate personal video, he or she may choose to use the video panel's "push" functionality to share the new video with the group.

This application is also designed to take advantage of Panelrama's automatic panel allocation feature. As mentioned in the previous section, the developer defines how important each device characteristic is for a panel (e.g., physical size is important for a video panel, and ready to hand is important for playback controls), and Panelrama assigns panels dynamically to a user's devices as each device or panel is added or removed.
For instance, a user who begins with all panels on a desktop computer may have the video panel automatically distributed to a newly connected television, followed by playback controls to a connected smartphone, leaving only search functions on the desktop. When a tablet is connected, all the playback and search controls migrate to the tablet, minimizing the number of devices a user needs to reach to interact with the video on the television. An example interaction sequence is shown in Figure 6.

FUTURE WORK
As a distributed interface framework, Panelrama lends itself well to the development of complex multi-user applications. Currently Panelrama does not differentiate between devices belonging to different users. We hope to investigate how to allow developers to define access user access permissions for individual panels, content restrictions for different users, and differentiation between owner and guest users.

In the future, we also intend to improve Panelrama to tackle security concerns in the distribution of panels. The allocation of panels to potentially compromised systems poses a security risk [1]. One approach would involve allowing developers to define content restrictions for public/private devices and panels.

Long term, we would integrate the ability to annex public devices. We will investigate hardware and software solutions to establish links dynamically between multiple devices, as well as implement adaptive interface designs to not only allocate the ideal panels for a device, but also modify the panels to use UI elements considered most usable on the device.

CONCLUSION
In this paper, we presented Panelrama, an HTML5-based distributed user interface framework combining UI distribution and automatic panel allocation. Panelrama gives developers the ability to create a single application across multiple devices. We also have a sample application: the video browser, which demonstrates the functionalities of the Panelrama framework. We believe that HTML-based frameworks, which minimize the work required for distribution of user interfaces, show great promise for the realization of distributed user interfaces.

REFERENCES