Distributed Physical Interfaces With Shared Phidgets

Nicolai Marquardt
Faculty of Media, Bauhaus-University Weimar
Bauhausstr. 11, 99423 Weimar, Germany
nicolai.marquardt@medien.uni-weimar.de

Saul Greenberg
Dept. Computer Science, University of Calgary
Calgary, Alberta CANADA T2N 1N4
saul.greenberg@ucalgary.ca

ABSTRACT
Tangible interfaces are best viewed as an interacting collection of remotely-located distributed hardware and software components. The problem is that current physical user interface toolkits do not normally offer distributed systems capabilities, leaving developers with extra burdens such as device discovery and management, low-level hardware access, and networking. Our solution is Shared Phidgets, a toolkit for rapidly prototyping ***distributed*** physical interfaces. It offers programmers 3 ways to access and control remotely-located hardware, and the ability to create abstract devices by transforming, aggregating and even simulating device capabilities. Network communication and low-level access to device hardware are handled transparently, regardless of device location.

ACM Classification: H.5.2 [Information Interfaces]: User interfaces – input devices and strategies, interaction styles, prototyping, user-centered design

Keywords: Distributed physical user interfaces, Phidgets.

INTRODUCTION
Physical user interfaces are increasingly important in many emerging visions of human computer interaction: ubiquitous and calm computing, tangible interfaces, pervasive and context-aware computing, information appliances, reactive environments, interactive art, ambient displays [21,7,17,19,4]. In most visions, physical user interfaces comprise an appliance constructed from simple hardware devices – sensors, switches, actuators, displays, motors, RFID – that developers package in some manner, and connect to, monitor, and control via software. These appliances are either carried by people or deployed at meaningful locations within the end user's everyday environment, with the idea that they work within (rather than apart from) the everyday practices of people [8].

While some physical user interfaces work as stand-alone appliances, they are usually envisaged as components interacting within a network of other devices [4,7,12,19,21]. For example, Weiser's Ubicomp vision anticipates "a network that ties [devices] all together" [21]. Dey et. al. explains context-aware applications: "…the devices used to sense context most likely are not attached to the same computer running an application that will react to that context" [7]. Similarly, Brave et al. extend tangible interfaces to distributed CSCW through the notion of "synchronized distributed physical objects" [4]. All perceive the system as a *distributed physical user interface* comprising various hardware devices connected to different computers over multiple locations, all networked together.

From this perspective, a distributed physical user interface is best viewed as hardware nodes on a distributed system. This viewpoint reveals that developers of such systems face two considerable challenges.

1. They must program, communicate with and control low-level hardware devices.
2. They must assume the additional programming burden inherent in most distributed systems: resource discovery, network communication and protocol development, connection control, managing failures due to connectivity problems and latency, debugging intricacies of distributed systems, and so on.

Toolkits are now available that simplify device programming (point 1), or that simplify distributed systems development (point 2), but not both. Consequently, our goal was to design a toolkit that lets programmers easily access the many distributed devices that comprised a network of physical user interfaces. Our solution is Shared Phidgets, a significant and powerful software extension of the commercial Phidgets platform [20]. Shared Phidgets:

- automatically discovers devices connected to a myriad of different computers;
- manages all network aspects so that no network programming is required;
- uses the same API to control a device, regardless of whether it is attached to a local or a remote computer;
- uses a *dMVC* distributed Model-View-Controller design pattern to represent devices so that data associated with the model is easily queried and manipulated;
- generates notifications across the network whenever device state is changed;
- offers graphical 'skins' that let a person view local and distant device state and control them via a GUI;
- offers the means to create or simulate 'abstract devices' that transform and aggregate low level hardware device capabilities into higher level abstractions; and

Marquardt, N. and Greenberg, S. (2007)
Distributed Physical Interfaces with Shared Phidgets. *Proc. 1st International Conference on Tangible and Embedded Interaction.* (Feb 15-17, Baton Rouge, Louisiana, USA). Also available through the ACM digital library.

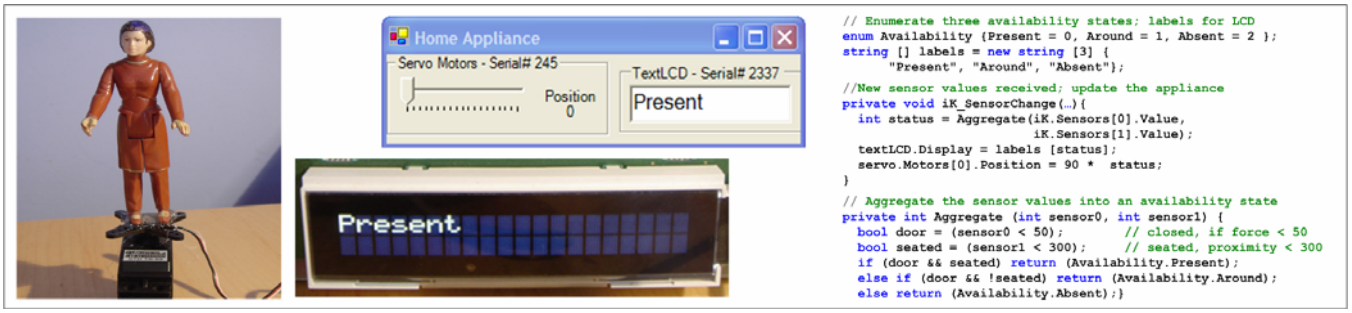

Figure 1. The home appliance and its graphical interface

- provides a set of high-level tools to: manage hardware and network connections, control, emulate and observe devices, and examine the dMVC model.

To forewarn the reader, this paper concentrates on the technical infrastructure underlying Shared Phidgets. Our belief is that the toolkit offerings and its architectural makeup comprise: (a) substantial software engineering contributions that tremendously simplify the development of distributed physical user interfaces, and (b) strong intellectual contributions by the way this architecture offers much more than a stapling of a physical user interface toolkit to a networking toolkit.

After summarizing related work, we use a 'hello world' program to show how a person would use Shared Phidgets to create a distributed awareness appliance. We then detail the Shared Phidgets architecture and illustrate the power of its advanced features.

RELATED WORK

Until recently, physical user interface development was restricted to the few programmers who knew about hardware and circuit design, and who were willing to do a huge amount of back-end programming: firmware, networking and protocol development to link hardware and computers, and application software that monitors and uses the device. Fortunately, various toolkits now makes rapid physical user interface development reachable by average programmers. Well-developed commercial offerings are Phidgets [20,13] and MakingThings, while research tools include iStuff [2], Input Configurator [9], d.tools [16] and Calder [18]. Offerings typically provide hardware devices with well-defined functionality, easy connectivity (including wireless) between device and a single traditional computer, and a software API that lets a programmer access the hardware functionality. Their power is that developers can focus on high level design of physical user interfaces rather than on low level implementation details [1,11].

Perhaps the most used of these toolkits is Phidgets, first created as a research system [13], and then commercialized by Phidgets Inc [20]. Phidgets includes USB-based hardware boards for input (e.g., temperature, movement, light intensity, RFID tags, switches) and output actuators (e.g., servo motors, LED indicators, LCD text displays). Its architecture and API lets programmers discover, observe and control all Phidgets connected to a single computer.

While all the above toolkits simplify hardware programming, they do nothing to help one manage hardware as a distributed system. Dey et. al.'s Context Toolkit [7] is the exception. Its *context widgets* abstract the actual (possibly distributed) devices and software used to collect contextual information. *Interpreters* transform this low level information into high level abstractions. *Aggregators* collect, group and logically relate multiple pieces of information. *Services* use the above input components to control something, i.e., to perform an output. *Discoverers* maintain a registry of components. Under the covers, components communicate through a TCP-based subscription-based event system.

Yet the Context Toolkit does not ease how programmers compose low level hardware devices. Our understanding is that a context widget's connection to hardware (including hardware control) has to be custom coded; the toolkit itself supplies no support for this difficult step. That is, the toolkit begins with the abstracted 'context widget' but does not explicitly support how these are linked to hardware. Thus there is a significant gap between how one accesses the hardware (as provided by the previously mentioned toolkits) *vs.* how one leverages this hardware in a distributed setting (as in the Context Toolkit). This gap is the 'sweet spot' that our Shared Phidgets toolkit addresses. As we will see, we extend the existing Phidgets architecture so that programmers can access low-level hardware devices located anywhere on the network, and compose them to work together in powerful ways.

A 'HELLO WORLD' PROGRAMMING SCENARIO

To set the scene, we illustrate how 'Jim' uses Shared Phidgets to create an awareness appliance that lets a person at home know if his working spouse is present, around, or absent from her office. The appliance comprises three linked devices distributed across two locations: the home and the office. While simplistic, it implements 3 previously published ideas: Door Mouse [4], Physical but Digital Surrogates [14], and Aggregates [7].

Description

The office part (not illustrated) comprises two off-the-shelf sensors attached to a Phidget InterfaceKit circuit board [20] plugged into the 'office' computer. A proximity sensor detects if someone is seated at the desk, while a force sensor detects if the office door is closed.

The home part, illustrated unadorned in Fig. 1, contains a Phidget TextLCD display, and a figurine atop a Phidget Servo [20], both plugged into a home computer. It also contains a graphical user interface (Fig. 1) mirroring the state of these devices. The program in Fig. 1 aggregates these two sensor values into a new 'availability' value:

- *present:* door open, someone seated;
- *around:* door open, no one is seated;
- *absent:* door closed, seated state ignored.

This program also adjusts the figurine's position and LCD display contents depending on this availability state:

- *present:* faces forward (0^o), says 'Present'
- *around:* faces sideways (90^o), says 'Around'
- *absent:* faces the wall (180^o), says 'Absent'.

Implementation

Shared Phidgets includes a run-time architecture. Jim starts a *Shared Phidgets Server* on a central computer of his choice (e.g., tcp://demo.ca:test), and a *Connector* on the local computer to link to this server. This takes seconds.

Jim first works on the office sensors. He positions the proximity sensor in front of the desk chair, tapes the force sensor to the inside of the door jamb, and plugs both into the Interface Kit. The *Connector* immediately detects the InterfaceKit as it is plugged into the office computer, and publishes its sensor data to the *Server*. This data is now available to other software connected to the *Server*.

Jim then builds the home part of this device illustrated in Figure 1. He glues the figurine to the motor, and packages that and the display into a box (not shown). He then writes a small program (Fig. 1 right) to monitor and aggregate the two distant sensors, and uses this aggregation to reposition the figure and determine the LCD display contents. Jim does this in 4 steps: he uses an interface builder for the first 3, and writes code only in the last step.

1. ***Connect to the server.*** Drag and drop the Shared Phidgets ConnectionManager object onto the window form, and set its SharedDictionary property to "tcp://demo.ca:test. This object automatically connects to the central server.
2. ***Create an object connected to the distant InterfaceKit.*** Drag and drop an InterfaceKit object onto the form. Set its SerialNumber property to the serial number of the distant InterfaceKit hardware board located in the work office. The software and distant hardware are now linked.
3. ***Create objects that control and graphical display the local appliance.*** Drag, drop and link a Servo and ServoSkin object, and a TextLCD and TextLCDSkin object into the form. Set their serial numbers to match the local hardware. These 'Skins' are graphical interfaces that reveals the state of the servo and the display hardware; while not strictly necessary, this graphical view is included for illustrative purposes (Fig. 1, middle top).
4. ***Monitor the sensor values to control the local appliance.*** Create an event handler for the interfaceKit's SensorChange event to monitor the current values of these sensors. Jim also creates a utility method Aggregate to aggregate the sensor values, and uses this aggregate to control the home appliance. Figure 1 shows this code.

He compiles and runs this program on the home computer. It automatically connects to the server over the internet, and raises events as sensor data collected from the office computer changes. The event handler code in Figure 1 is invoked and the home appliance is adjusted accordingly.

Discussion

The above example is notable in that its programming is almost identical to that of the original non-distributed Phidgets [13] – but in the original all devices needs to be connected to one machine. Aside from starting the server and the Connector programs, the only coding difference is that the programmer included a ConnectionManager and an address to the server. All distributed systems aspects are otherwise hidden. We stress that this example only shows the most basic use of Shared Phidgets; much more sophisticated and nuanced distributed appliance designs are possible, as illustrated in later sections (e.g., aggregation can be done separately, as in [7]).

SHARED PHIDGETS ARCHITECTURE

We now concentrate on what happens 'under the covers'. Fig. 2 illustrates the Shared Phidget hardware, runtime and programming architectural components, its programming libraries, and the interactions between them. Subsequent sections describe what application developers actually see and the tools they use to facilitate the programming process.

Hardware and Devices

Hardware devices are the combined hardware/circuit board building blocks primarily exploited by our infrastructure (Fig. 2b). Developers use these devices to create the physical portion of their interface, which in turn defines the end user's interaction (Fig. 2a). The Shared Phidgets architecture provides access to all Phidgets Inc. hardware devices [13,20], although non-phidget devices can be integrated as well. These include *input sensing* (motion, touch, proximity, light…), *manual input controls* (switches, dials, sliders, joy sticks, key fobs, RFID readers…), *output actuators* **(**motors, servos, solenoids) and output *displays* (lights, text displays…) (Fig. 2b). Each connected device is uniquely identified by its location, type and serial number.

Computer Communication to Phidget Devices

Phidget devices interact with a controlling host computer, and thus need to be connected to them. Currently, all Phidget devices connect to a host computer via USB.

Phidgets Inc. supplies two rudimentary interfaces to let programmers communicate with these devices. First, a dynamic link library API offers access all locally attached devices. Second, a 'web service' provides a socket-based interface to the local machine's Phidgets (Fig. 2c). While this second form can be exploited as a crude network service, Phidget Inc supplies it primarily as a platform-

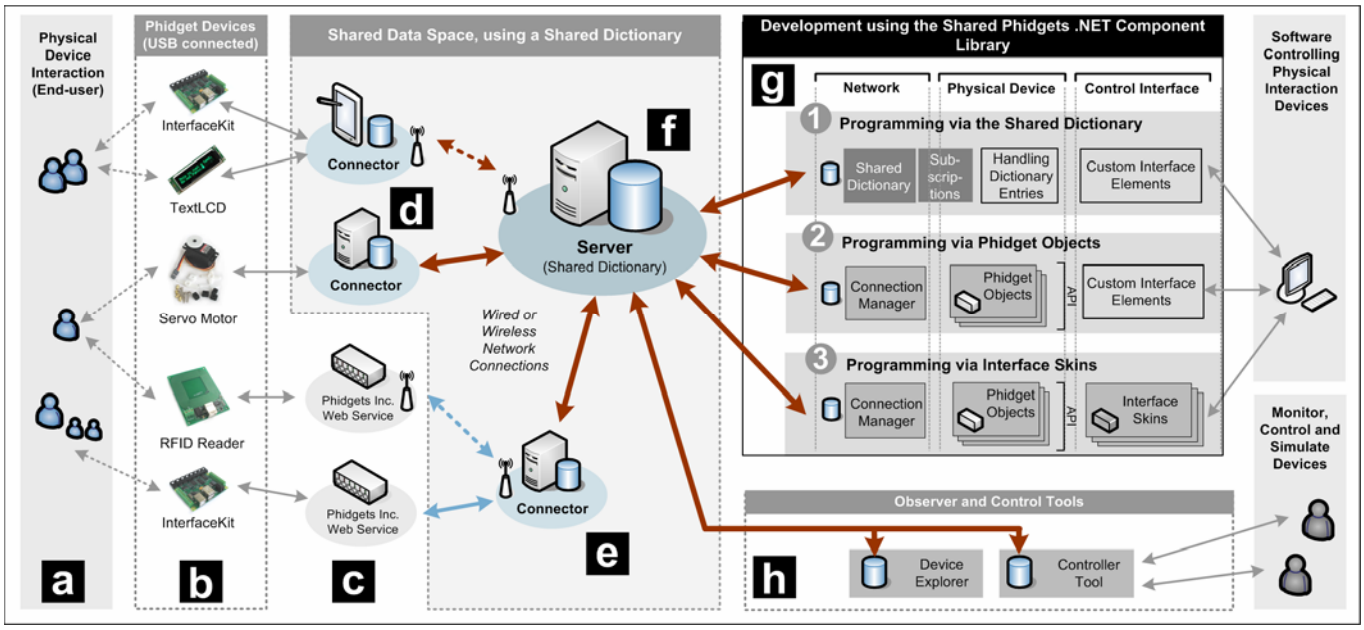

Figure 2. An overview of the Shared Phidgets Architecture

independent interface to simplify access to Phidgets across different programming languages and operating systems.

So far, we have described the offerings of Phidget Inc., which closely matches the original Phidget architecture [13]. Remaining sections depart radically from Phidget Inc. offerings. As we will see, our new distributed architecture provides a *shared distributed data space* that contains information about all phidget devices regardless of their location, and a *connector* mechanism that hooks devices on the local computer to this shared data structure.

Shared Distributed Data Space

A fundamental component of the Shared Phidgets architecture is a shared data space. We implement this as a distributed data structure using the *shared dictionary* provided by the GroupLab .NETWORKING toolkit [3].

.NETWORKING allows client processes read / write access to a collection of data objects maintained within the *Shared Dictionary Server* (Fig. 2f). This is also a notification server, as clients are immediately notified of any changes to data they are subscribed to, regardless of who made these changes. .NETWORKING also manages all runtime networking housekeeping tasks: socket creation/teardown, wire protocol, data marshalling, parsing, etc.

Data in the shared dictionary is structured as hierarchical key/value pairs. Values can be primitive data (integers, strings…), binary data (images…) or complex data (lists, structures…). A key is expressed as a path hierarchy. A rich set of operators allow programmers to subscribe and iterate over data held in particular sub-paths of this hierarchy[3].

Shared Phidgets leverages the shared dictionary by managing data and the ways participating machines access this data as a *distributed Model-View-Controller* (dMVC) pattern [15]. The *model* is the abstract data stored on the shared dictionary. Multiple *controllers* – the client machines – can change values in the shared data model. As multiple clients receive notifications of changes of that data, they can each generate their own *view* of it.

For every phidget seen, Shared Phidgets automatically creates a model entry that completely defines the phidget state. Using our office/home appliance example, Fig. 3 shows the partial dictionary entry for its three Phidgets: the Servo, InterfaceKit and TextLCD. To illustrate the dictionary's hierarchical nature, consider the key \sharedphidgets\phidgetservo\418\servoposition\0. Its path specifies the root (Shared Phidgets), the device type (Phidget Servo), its unique serial number (418), and that it contains the servo position attribute of motor #0 (currently 90^{O}). Thus given a serial number of a Phidget Servo, it is easy to search for it, and to read, modify or iterate through all its properties and values. For the InterfaceKit, we see the values of sensor 0 (the door pressure sensor) and sensor 1 (the proximity sensor). We also see that the TextLCD is currently displaying the text "Present". As a person triggers the office sensors (the controllers), the model is updated and a notification raised that invokes the callback in Fig. 1 in the distant home client. This updates the appliance's view by resetting the servo and text display values.

The critical point is that the Shared Phidgets architecture is realized primarily as a dMVC pattern over a distributed client / server shared data model with view updates triggered via a notification server. This greatly simplifies the internals of distributed data management and, as we will see later, provides three powerful ways for the developer to program Shared Phidgets.

Connectors and Phidget Proxies

The shared data space maintains a runtime model (representation of properties and current status) of all

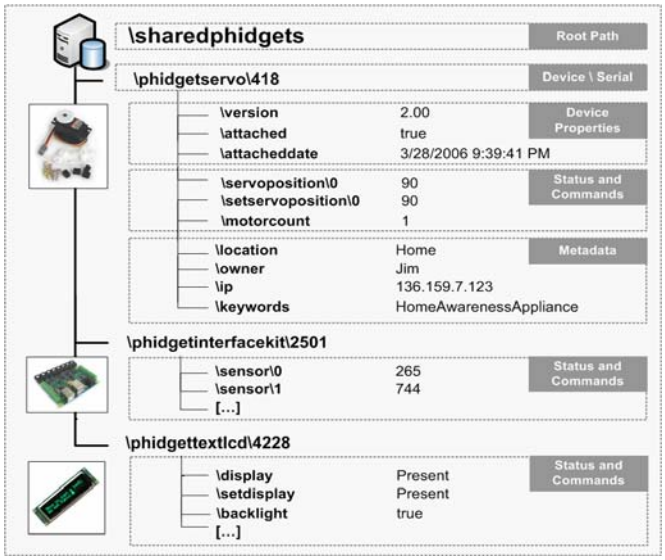

Figure 3. Path structure in the Shared Dictionary

distributed Phidgets, but does not define how a machine's Phidgets connect to it. This is the job of the *Connector*, illustrated in Figs. 2d+e. The Connector runs quietly in the background on each local client machine. It notices any locally-connected Phidgets that are plugged in over time. As it finds a new device, the Connector dynamically adds an appropriate *phidget proxy object* to handle it.

This proxy object has two responsibilities. First, it observes and controls its specific phidget device features, e.g. an interface kit object observes all sensor values generated by the hardware; a servo object controls a phidget Servo position. Second, it serves as an intermediary between the phidget device and the shared dictionary model: it ensures that device attributes and matching key/value pairs crated in the data model reflect the same state. Acting as a controller, it monitors the phidget device for any changes, and updates the shared dictionary model to reflect those. For example, a reported sensor value will update its corresponding data model entry. At the same time it acts as a view, where it monitors the shared dictionary model (via notifications) for any data updates, and adjusts the phidget device to reflect that new value. For example, a change in a servo position data will translate to the servo motor actually rotating to that position. We mentioned that non-phidget hardware can be included in architecture; this is done by creating a proxy specific to that hardware to interact with the dMVC.

Revisiting Fig. 3, we now see that it illustrates how 3 proxies have modeled three connected phidget devices in the shared dictionary; excluding timing delays, these represent the properties and current state of the hardware as attributes. Attributes describe three categories.

General device properties provide information common to all Phidgets: the h/w version number, whether this device is currently attached to the local computer, and when it was.

Current device status represents the available input and output functions of the phidget. Each sensor or control input involves a separate dictionary entry, and the Connector tool is responsible for controlling (updating) these values as the hardware triggers updates to them as events. An example is illustrated by the values shown for the sensor inputs of the InterfaceKit. Each actuator or display output is represented by two entries: one with the current value of the output, and one for submitting requests to change the value (e.g., the servo's servoposition vs. setservoposition in Fig. 3). These two entries are important: the set entry is what the output should be, and this is in turn used by the Connector to direct the hardware until that value is achieved in the corresponding entry (depending on the device and network, this could be near-instantaneous or take several seconds).

Metadata entries contain additional information entered by the proxy that describes each device. As seen in Fig. 3, this includes the IP address of the computer a phidget is connected to, its physical location, its owner, and keywords. The actual metadata is specified through the local machine's Connector interface. While some metadata fields are automatically provided, the end user can fill in a form within the Connector to add one or more custom fields and values, and associate these with a particular phidget device. These custom metadata entries are then stored within that phidget's data model in the shared dictionary. For example, the three Phidgets comprising the Fig. 1 appliance may be viewed as a single appliance by creating a metadata key 'ApplianceType' and setting its value to 'Jims Awareness Appliance'. Another metadata key 'Where' can indicate if it is the office or home side by setting its value to 'Office' or 'Home'. Because metadata information originates in the local machine's Connector, the metadata information is updated if and when users move devices between computers (e.g., new location, owner…). That is, the metadata information can offer context-dependant information that can be exploited by the programmer.

Internally, the Connector accesses either the Phidgets Inc. DLL (as in Fig. 2d), or it opens connections to one or more Phidgets Inc. web services (Fig. 2e). Through this web service connection, our Connector can serve as an intermediary to other computers hosting standard Phidget devices but not running Shared Phidgets software (Fig. 2e). This is important for it gives platform-independence: while Shared Phidgets now runs only on Windows, we can still connect to other Apple and Linux boxes hosting Phidgets.

For security, the Connector lets users activate password protection and encryption of transmitted data. If they don't want to share hardware, they can also exclude connected Phidgets devices from the shared data space.

In summary, the Connector mediates between the data *model* and the phidget hardware discovered on the local machine or through web services. As a *view* onto the model, the Connector commands physical hardware to reflect data state changes made from client applications. As a *controller*, it transforms state information of the physical widget into changes to the data. As a local data store, it

adds *metadata* that identifies particular features of that device as it relates to its local installation or its intended use. Finally, all this happens without any user or programmer intervention. The Connector can automatically start on login and run in the background. End users can optionally raise a *Connector GUI* to monitor the status of all shared devices, and to add and/or alter the metadata.

DEVELOPMENT STRATEGIES - PART 1

The dMVC model considerably simplifies the job of sharing and manipulating Phidget devices across a network and across multiple machines. Yet we recognize that the dMVC pattern may be unfamiliar to average programmers. Consequently, our architecture offers a three-tier programming model (Fig. 2g) that lets developers choose and intermix 3 different development strategies that balances power *and* simplicity. All strategies work within the standard Visual Studio development environment and C# programming language. All strategies also trivialize hardware access and network communication, thus letting developers focus on their physical interface design ideas.

Strategy 1: Programming via the Shared Dictionary

Programmers can develop Phidget applications by addressing the shared dictionary directly (Fig. 2g, row 1). While this adds power, programmers need to know the API to the .NETWORKING shared dictionary (e.g., subscription management and data organization), regular expressions (for pattern matching), the Shared Phidgets specification for device representation, and be familiar with the dMVC pattern. With subscription objects, developers receive notification events of changes to entries in a sub-tree of the dictionary hierarchy. They also write custom code as event handlers that take action on changes to dictionary entries.

This programming model is very powerful as developers profit from pattern matching via path wildcards. That is, they can easily iterate through and control many devices at the same time. For example, the single line below returns all motors on all Phidget devices on all computers, where the '?' is a regular expression that generates matches for a single hierarchical level.

```
/sharedphidgets/phidgetservo/?/setservoposition/?/
```

By embedding this line in a foreach statement, a programmer can iterate over this expression to (say) read all motor positions of all Phidget Servos, or reset them to 180^{o}.

The dictionary can also be used to create new *abstract devices* that monitor, transform, and aggregating low level hardware device values into new data entries. As an example, reconsider our office/home appliance of Figure 1. In that system, the home component created and used the Availability aggregate. In practice, this should be done by the office component; this would let us (say) add new sensors to more accurately infer presence without changing the home side of the appliance. Similar to the code in Fig. 1 the office side could calculate availability state, but store this aggregated value into a new shared dictionary SD entry that models an abstract device called 'officepresence':

```
SD ("/sharedphidgets/officepresence/1/availability") = "Present"
```

The home appliance can then subscribe to this abstract device instead of the InterfaceKit, where it monitors its value to control the figurine position and LCD display.

Strategy 2: Programming via Phidget Objects

In cases where the power of the Shared Dictionary is not required, the programmer can develop Shared Phidget applications by a simpler object-oriented API that completely encapsulates individual device capabilities. This is what was done for coding the home part of the appliance of Fig. 1. We had mentioned that the Connection Manager provides a *phidget proxy object* matching each phidget device (Fig. 2g row 2). This proxy reads and modifies that Phidget's entries in the shared dictionary according to a particular device, and provides an object-oriented properties/methods API for the developer. A programmer controls a device by altering its properties and/or invoking its methods, and monitors changes to device status by adding event handlers. Networking and distributed data sharing aspects are completely hidden. Not only is this simpler than directly accessing the shared dictionary, but this familiar programming model requires little extra learning as it matches conventional GUI object-oriented programming and the original Phidgets programming paradigm [13]. Unlike the original Phidgets [13], the actual devices may now be located anywhere on the network; the programmer simply links the Shared Phidget object to the distant device by specifying the hosting computer's metadata location and/or the device serial number.

Strategy 3: Programming via Interface Skins

Our simplest strategy lets developers compose an interface by dragging and dropping GUI representation of phidget devices, where these GUIs let end users monitor and control devices regardless of their location. While these GUIs can be programmed from scratch atop of phidget objects, programmers will typically use *interface skins*.

Interface skins are wrapper objects around the phidget proxy object API. They normally provide GUI representations for every sensor or actuator functionality of the corresponding Phidget device (Fig. 2g, 3rd row). Using an interface builder, developers simply drag and drop these skins into a window, as done with our home appliance. In our experiences, skins provide an intuitive starting point for developers who have never dealt with hardware, where it entices them to experiment with the variety of physical interaction devices. It also serves as a very effective debugging mechanism.

We call them 'skins' as a single Phidget device can be represented by multiple GUIs (this differs from the strategy first described in [13]). For example, the Interface Kit can have a skin that displays all sensor values in a textual table, as animated sliders, or as a graph that shows changes over time. Programmers can also create their own custom skins

(using an included set of abstract base classes) to visualize a device in any way they wish. They can also create a skin to an abstract phidget that aggregates properties collected from several devices (e.g., as described in the shared dictionary subsection). A programmer can even create multiple views onto phidget devices by connecting multiple skins to one or more devices.

DEVELOPMENT STRATEGIES – PART 2

Three other capabilities complete our description of Shared Phidgets: metadata, extensibility, and tools.

Programming using Metadata and Device Discovery

Metadata is an important extension to the Phidgets concept as it allows people to attach context and device-specific information to devices, and to discover devices that match. Metadata is accessed via the shared dictionary or the Shared Phidget object in the same way as other Phidgets properties. For example, one accesses the servo's location and keywords in Fig. 3 through the servo object properties:

```
String location  = servo.Location;
String keywords = servo.Keywords;
```

Programmers can also use metadata to discover devices across the network. The easiest way is through filtering: filter terms are added to a Phidget object, and the device whose metadata matches those terms are attached to it. For example, the following code will discover the InterfaceKit used in our example and in Figure 3 by matching its Location and its Owner properties (other filter properties include serial numbers and IP addresses):

```
interfaceKit.FilterLocations.Add("Home");
interfaceKit.FilterOwner.Add("Jim");
```

Custom metadata can be accessed, exploited and even reset via a hashtable associated with a Phidget Object. Programmers can iterate over all its metadata entries to look for matches, or see if a particular one exists through a ContainsKey() method, or get a particular value through the GetMetadata("key") method. For example, a programmer can unify a set of devices into an appliance simply by setting a common metadata tag that is later matched. One can also add metadata at runtime. For example, if a programmer had access to a servo's GPS location, it can be added on the fly:

```
servo.AddMetadata("GPS","N-51-02:W-114-01");
```

This user-defined metadata is immediately visible in the Connector tool, and other connected applications can discover it by iterating over the metadata collection.

Extensibility

Shared Phidgets is an extendable architecture. It is straightforward to include new phidget devices as they are made available by Phidgets, Inc. New interface skins can be created for existing Phidgets, for abstract Phidgets, and for new hardware. The toolkit offers abstract base classes as the building block for these phidget proxy classes and the interface skins; all implemented device objects and skins are derived from these. The base classes provide the main API and implementation of the phidget discovery methods, shared dictionary connection, and subscription objects. Devices are not limited to Phidgets, and hardware from other vendors can be included, albeit with modest effort.

Observer and Controller Tools

Shared Phidgets comes with two important tools, each constructed atop the programming environment, that lets a developer or end-user monitor, control and even simulate all devices distributed across the network (Fig. 2h).

The *Dictionary Explorer* provides a direct view into the dMVC data model, and is invaluable for debugging. The user can observe, modify, and create entries in the data space, and they can also create simulated devices by adding needed set of shared dictionary entries that specify the device type and its properties.

The *Device Explorer* provides a compact and more readable overview of all devices across the network, including their serial number, attached status, metadata such as location, owner, etc. The user also has the option of selecting a device, which automatically creates its interactive skin: they can then monitor and/or adjust that device's values.

EXAMPLES

A few distributed physical user interfaces are listed below to illustrate the power of the Shared Phidgets toolkit.

LumiTouch comprises a pair of interactive picture frames [6]. When a person touches one frame, the other frame is lit. This Tangible Media Group project is trivially replicated with two InterfaceKits, one per frame, each with attached LEDs and touch sensors. The lights on one frame are controlled by simply monitoring the touch sensor state on its distant partner frame.

Location-based messaging. Elliot et. al designed a system that allowed people to send notes to devices at particular locations within the home [10]. Their work used standard Phidgets, so much coding was needed to configure these distributed devices. This is far easier in Shared Phidgets. Each device (e.g., a Phidget TextLCD Display) is tagged with location metadata, e.g., 'Kitchen' or 'Hallway'. When a person wants to send a note, the underlying program queries and finds all the location metadata on the distributed system, composes this as a popup menu, and then 'sends' the message to the chosen location by setting the text property of the TextLCD Display at that location.

Sensor Maps creates a map overview of all hardware and abstract devices held in the dMVC (any existing map image can be loaded). Attached devices are displayed as small circles atop the map, and the occurring events are displayed as expanding/fading circles (e.g., the radius depends on the current sensor value being tracked). Clicking a circle raises its skin. The inset

illustrates distributed sensors across a university campus. Similar maps could let a person control the state of all appliances located within a smart home.

DISCUSSION

Shared Phidgets contribute a significant *software engineering contribution* to tangible computing by the way its architecture cleanly combines networking power with physical hardware. The result is a robust, easy to use, and very practical toolkit and run-time system. More importantly, the power of the Shared Phidgets architecture offers *significant intellectual contributions* at five levels:

Distributed device access. Our dMVC approach adds considerably to how programmers interact with distributed hardware devices, something not done by other toolkits.

Programming simplicity and power. Our architecture provides a 3-tiered way to program phidget functionality, which effectively trades off simplicity and power.

Semantic metadata. The ability to add metadata to devices means that programmers can leverage semantic meaning associated with each device and location. Expected uses include programmatically deciding location and context-dependant actions, and to discover device groupings so they can be collectively considered as an appliance.

High level tools built atop our dMVC model lets end users visualize and control devices across the distributed network, and even add simulated devices. This is also tremendously useful for debugging as they can probe current state.

Abstract devices. Finally, we stress that the ability to create new abstract devices from a combination of hardware building blocks or through simulated devices is extremely powerful. Letting people define abstract devices is the key to how we bridge between traditional physical user interfaces and Dey's context widget. The programmer can also create new API's and Interface Skins that further wrap these abstract devices, i.e., as particular appliances with APIs and GUIS that reflect its semantics and appearance.

CONCLUSIONS

Shared Phidgets is a new generation physical user interface toolkit. It recognizes that many physical user interfaces will comprise interacting distributed components, and that these components will be remixed in a variety of ways.

In the past, programmers were responsible for all distributed systems aspects. The Shared Phidgets dMVC architecture takes over this chore. Surprisingly, there is almost no extra programming penalty, unless one wants the extra power offered by directly accessing the dMVC.

There are other significant features: Interface Skins serve as both views and controllers into the distributed model, and new skins can be created to match end uses. Through metadata, people can add device-specific, location and context-dependant attributes to the appliances they create. Finally, the capabilities of Phidgets can be interpreted, recombined, aggregated, simulated and abstracted into abstract devices, which bridge into the powers offered by the Context Toolkit [7]. These abstract devices can be further enhanced as programmable objects and skins.

Acknowledgements. Funding by NSERC NECTAR Smart Technologies, Inc. and Microsoft. Thanks to Phidgets, Inc.

Software & tutorials: grouplab.cpsc.ucalgary.ca/cookbook

REFERENCES

1. Abowd, G. D. Software engineering issues for ubiquitous computing. *Proc IEE Int'l Conf. Software Engineering*, 1999.
2. Ballagas, R., Ringel, M., Stone, M., and Borchers, J. iStuff: a physical user interface toolkit for ubiquitous computing environments. *Proc ACM CHI*, 2003.
3. Boyle, M. and Greenberg, S. Rapidly Prototyping Multimedia Groupware. *Proc DMS*, Knowledge Systems Institute, 2005.
4. Brave, S., Ishii, H. and Dahley, A. Tangible interfaces for remote collaboration and communication. *Proc. ACM CSCW*, 1998. 169-178.
5. Buxton, W. Living in augmented reality: Ubiquitous media and reactive environments. In Finn, Sellen & Wilber (Eds.). *Video Mediated Communication*, Erlbaum, 1997. 363-384.
6. Chang, A., Resner, B., Koerner, B., Wang, X, and Ishii, H. LumiTouch: An emotional communication device. *ACM CHI Extended Abstracts*, 2001. 313-314.
7. Dey, A. K., Salber, D. & Abowd, G. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-Computer Interaction*, 16, 2001.
8. Dourish, P. *Where the action is: The foundation of embodied interaction*. MIT Press. 2001.
9. Dragicevic, P. and Fekete, J. The Input Configurator toolkit: towards high input adaptability in interactive applications. *Proc. AVI*, ACM Press, 2004. 244-247.
10. Elliot, K., Neustaedter, C. and Greenberg, S. StickySpots: Using Location to Embed Technology in the Social Practices of the Home. *Proc Tangible and Embedded Interaction*, 2007.
11. Greenberg, S. (2007) Toolkits and Interface Creativity, *J. Multimedia Tools and Applications*, 32(2), Kluwer.
12. Greenberg, S. Collaborative physical user interfaces. In K. Okada, T. Hoshi and T. Inoue (Eds) *Communication and Collaboration Support Systems*. IOS Press, 2005.
13. Greenberg, S. and Fitchett, C. Phidgets: Easy development of physical interfaces through physical widgets. *Proc ACM UIST*, 2001, 209-218.
14. Greenberg, S. and Kuzuoka, H. Using digital but physical surrogates to mediate awareness, Communication and privacy in media Spaces. *Pers. Technologies*, 4 (1), Elsevier, 2001.
15. Greenberg, S. and Roseman, M. Groupware toolkits for synchronous work. In M. Beaudouin-Lafon (Ed), *Computer-Supported Cooperative Work (Trends in SW 7)*, Wiley, 1999.
16. Hartmann, B., Klemmer, S., Bernstein, M., Abdulla, L., Burr, B., Mosher, A. & Gee, J. Reflective physical prototyping through integrated design, test, and analysis. *Proc UIST 2006.*
17. Ishii, H. and Ullmer, B. Tangible bits: Towards seamless interfaces between people, bits and atoms *Proc ACM CHI* 1997
18. Lee, J., Avrahami, D., Hudson, S., Forlizzi, J., Dietz, P. and Leigh, D. Calder toolkit: Wired and wireless components for rapidly prototyping interactive devices. *Proc ACM DIS*, 2004.
19. Norman, D. *The Invisible Computer.* MIT Press, 1998.
20. Phidgets, Inc. www.phidgets.com. April. 2006.
21. Weiser, M. The Computer for the 21st Century. *Scientific American.* 94-110, September, 1991