Unpublished manuscript

A Toolkit to Support Rapid Construction of Ubicomp Environments

Chris Greenhalgh, Shahram Izadi, Ian Taylor, James Mathrick

School of Computer Science and IT, University of Nottingham, Jubilee Campus, Wollaton Road, Nottingham, NG8 1BB, UK
{cmg, sxi, imt, jym}@cs.nott.ac.uk

Abstract. We describe the design and initial implementation of a toolkit for ubiquitous computing that reduces the cost – especially in time and effort – of developing applications and environments, and increases the potential involvement of designers and users in this process. The toolkit supports distributed applications running over multiple hosts by the creation, configuration and interconnection of software components (both toolkit-aware and existing components) and components which represent physical devices and sensors. Users are supported by tools which provide various representations of the running environment, plus facilities for scripting and learning by example.

1 Introduction

Over the last three-to-four years we have been involved in a large multi-site initiative to explore the development, deployment and use of ubiquitous computing environments. This has involved the realization of a number of distinct user experiences including:

- The development of reactive learning spaces that exploit ubiquitous computing technologies to promote a sense of exploration within children [1].
- The construction of reactive museum spaces that use ubiquitous computing devices to enhance the users' visiting experience [2].
- The development and placement of technologies within different domestic settings [3].

The development of these experiences has involved the assembly of a heterogeneous collection of devices (and platforms), the placement of these devices within a given space and the development of facilities that allow these devices to interconnect in a variety of different ways. This process has also been adopted for the realization of a number of "outdoor" experiences including:

- The augmenting of a physical wood with reactive technologies to promote learning [4].
- The construction of a series of mobile games that link on-line users with "on the street" players [5].
- The extension of museum visiting to consider the support for visitors exploring a city [6].

The development of these different experiences has proven to be a costly endeavor with considerable effort being required to allow the construction of these reactive ubiquitous environments. Our experiences have shown that:

- We have needed to integrate across a number of different hardware and software platforms.
- We have needed to integrate the developed reactive space with existing facilities outside our control (for example museum catalogues or web based services for online games).

In order to do this we have exploited a number of lightweight communication fabrics (e.g. ELVIN [10]) that allow a disparate set of devices to talk to each other. However we have found the development to be a long term endeavor with considerable development resources going into determining how best to interconnect devices, often at the cost of construction of new ubiquitous devices or the exploration of different possibilities for the arrangement of devices.

Given our overall aim of exploring the deployment and use of ubiquitous experiences as a means of driving our research forward, we wish to significantly reduce the cost of constructing spaces that allow the deployment of ubiquitous devices and provide facilities which make the exploration of different combinations of devices as low cost as possible, especially in terms of human resources. This paper presents the development of a toolkit and associated component framework that has allowed us to significantly reduce the cost of this development.

2 The importance of toolkits

The use of software toolkits has proven popular in the development of interactive systems [7, 8]. Their ability to reduce the cost of application development (through code reuse and modularity) has allowed researchers to rapidly prototype and experiment with interactive systems. The emergence of these toolkits has increased the involvement of designers and users in the prototyping process [9] and empowered designer and users to develop customized applications [20].

Our challenge is how to realize an equivalent set of toolkits for ubiquitous computing that reduce the cost of development and increase the involvement of designers and users in the process. We are not alone in exploring the possible benefits to be gained from developing toolkits to support the rapid construction of ubiquitous applications.

Generic middleware platforms such as Jini [25], and the .NET combination of UDDI [29], WSDL [30], and SOAP [31] can lend themselves to these types of environments by allowing distributed devices and software applications to discover and communicate with one another over defined protocols. The context toolkit [11] represents an early example of a toolkit specifically developed for ubiquitous computing environments. It adopts a widget-based approach inspired from previous work within interactive systems. The toolkit demonstrates the advantages of a component based approach in developing context-aware applications for such environments. The CoolTown [32] project at HP labs is another example of an infrastructure for pervasive

computing. It builds upon the ubiquity of web technology, to provide access to components through URIs and handles communication through HTTP and HTML.

The development of these application toolkits has been complemented by the construction of hardware component based toolkits such as Phidgets [12, 13], Smart-Its [14] and Motes[15] that provide a generic set of hardware devices that can be specialized to particular applications. This work has reduced the cost of hardware development by providing a simple set of generic, stable and agreed hardware interfaces and protocols.

These different forms of platforms and toolkits reduce the cost of hardware and software development. In order to achieve this they require developers to adopt a particular approach and consider their development essentially from within that particular platform or toolkit. However, the development of ubiquitous computing environments often requires a number of different platforms and approaches to co-exist. The need to support heterogeneous approaches has seen a number of different researchers explore how best to consider the combination of different devices and toolkit services. This includes the work on iStuff [16] and the event heap [17] undertaken as part of the iRoom project [18] and the work on recombinant computing in the Speakeasy project [19]. The approach we have adopted builds directly on this work by considering how a number of different types of ubiquitous device (each with its own particular approach) can be represented in such a way that they can be easily combined in a language-independent manner.

Other work has seen the emergence of interactive toolkits that attempt to lower existing barriers of application development and allow designers and end-users to be more closely involved in the overall development process. Examples of this include scripting languages such as those found in Macromedia Director [20]. The development of these scripting languages has been complemented by learning by example approaches [21] that allow users to "train" interactive systems.

A similar strategy is evident within those developing ubiquitous computing toolkits. For example, the work on iCap [22] provides a user oriented programming interface. While Speakeasy [28] provides a generic browser application to allow end-users to configure and construct ubiquitous computing environment. The work on Accord [23] presents a simple jigsaw based interface to allow end users to assemble different arrangement of ubiquitous devices and services. This editor-based approach has been complemented by learning approaches that allow end-users to build ubiquitous applications by demonstration [24].

The key to our approach has been the development of a lightweight component toolkit which allows a number of different hardware and software systems to be interconnected and rapidly configured. A key driver for us is supporting dynamic ubiquitous environments where the arrangement of devices can be continually revised to meet the needs of users. Consequently, we have developed a general component framework to allow us to manage the lifecycle of components – from their initial instantiation within the space to their eventual demise. In the rest of this paper we present our general toolkit model, its current implementation, our experiences of developing trial applications with the toolkit and the future challenges arising from this work.

3 Toolkit Model

The toolkit considers a ubicomp application or experience to be realized as a dynamically interconnected and potentially time-varying collection of hardware and software components, distributed across a number of machines. This section describes the deployment and coordination model supported by the toolkit. The current version of the toolkit presumes a relatively self-contained deployment setting, such as a single home, with a wired and/or wireless local area network.

3.1 Locales

Each deployment setting (or "Locale") – such as a home – has a single Locale Master process. In a turn-key installation this would be running as a pre-configured service on a dedicated Locale Master machine or host, which might also act as a local DHCP server, wireless access point and internet gateway/router. The Locale Master process creates a shared data-space which is used for coordination and communication between the hosts participating in that Locale. The Locale Master also uses a multicast network discovery protocol similar to Jini's [25] to advertise the coordination data-space's existence on the local network.

Any host on the local network can then join the Locale by running a suitable Component Container process. Again, in a turn-key installation, this would be a pre-configured service running on each Locale Client machine or host. Therefore, when using a wireless (IEEE802.11) network, a new client machine could be added to the Locale simply by configuring its wireless networking (network name and any required security parameters).

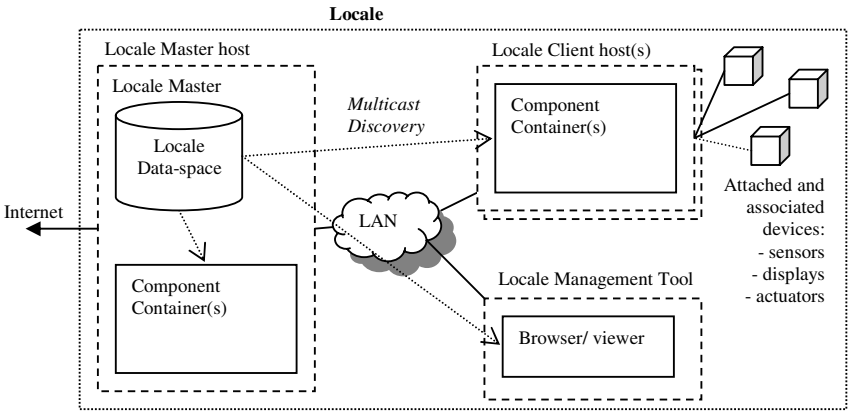

Fig. 1. Top-level Locale deployment model

In addition to normal Locale clients, various Locale Management Tools can also (usually temporarily) join the Locale, allowing interactive monitoring, troubleshooting and explicit configuration of the various aspects of the Locale. See figure 1.

3.2 Hosts and Containers

Varying numbers of Locale Client hosts can participate in a Locale. The various physical elements – sensors, displays and actuators – of the system would then be physically (or wirelessly) connected to these hosts. Each host can run any number of Component Container processes. Each Container process runs within the context of the host operating system and provides an environment within which toolkit software Components can exist and be managed. On start-up, each Container will discover the Locale data-space, and then uses this for distributed communication and coordination.

3.3 Components and Capabilities

The main purpose of a Container process is to allow the creation, management and coordination of software components on that particular host. These software components need not be written with any specific knowledge of the toolkit, since language reflection facilities can be used to host arbitrary software components (e.g. JavaBeans [26] in Java or CLR objects [27] in C#). However components can also make use of toolkit APIs to – for example – to become actively involved in managing and reconfiguring the Locale.

Our experiences in the development of a range of different ubiquitous computing environment suggest a number of typical kinds of software component:

- *Device driver interfaces* for physical extension points on the host, e.g. USB port, COM port;
- *Software proxies* for locally attached or associated physical devices, e.g. individual Phidgets [13] or SmartITs [14];
- *Software-only services*, e.g. a media viewer application, or interfaces to information repositories;
- *Application behavior and "glue"* components, e.g. scripts or learning/mapping components.

Some containers may be preconfigured to create certain Components on start-up, for example components to manage physical extension points such as USB ports. In other cases components may be created and destroyed directly by the Container or by already-active Components, for example when a new Phidget is connected to or disconnected from the host. In other cases the components will only be created in response to requests received via the data-space from other components or applications, for example a script component required as part of a particular application.

In any case the Container will advertise its ability to create Components of a certain type by maintaining a "Capability" data-item in the Locale data-space. This allows other applications and components – such as Locale Management Tools – to inspect the Locale data-space and determine (a) which Containers are currently available within the Locale and (b) what kinds of Components those Containers can create and manage (see figure 2(1)).

3.4 Request-based Component Life-cycle Management

The Locale data-space provides strong support for the sharing of stateful information, i.e. data-items. Any client of the data-space can add data-items to the data-space, and other clients can observe these – and their modification and removal – via a pattern-matching observer interface. Consequently, the data-space decouples information producers and consumers, both in space – they may be in different processes on different hosts – and in time – the data-item may remain in the data-space for an extended period of time, remaining available to observation and matching. The data-space also allows system inspection for a range of purposes including debugging, monitoring and dynamic adaptation, as well as for anticipated data exchange patterns.

To make the most of this facility the toolkit adopts a common idiom for communicating and representing requests and responses within the data-space:

- Each request (for example the request for a particular Container to create a particular Component) is represented by a data-item in the data-space, which is maintained there for as long as the requesting component maintains an interest in that Component (see figure 2(2));
- Similarly, each response (for example an advert indicating that a particular component now exists and is active) is represented by another data-item in the data-space, which is also maintained for as long as the thing to which it corresponds exists (see figure 2(3)).

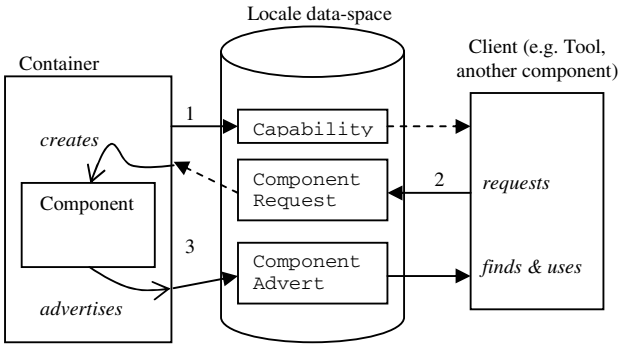

Fig. 2. Request-based Component Life-cycle Management

Note in particular that the response persists for as long as the Component exists; consequently a late joiner to the Locale can easily find existing Components. Similarly, the request persists until the Component is no longer required (not just until the component is created), so that a late joining container or a crashed and re-started container can easily find existing requests which it may be able to satisfy, and also so that the withdrawal of the request(s) can be used to determine that the Component is no longer required.

3.5 Component Model

A Component is the basic unit of deployment, management and coordination in the toolkit. A Component is a well-defined unit of functionality, which has a well-defined interface to external functionality – either that it requires or that it provides. The abstract Component model that we have chosen is compatible with the JavaBeans [26] model and COM/CLR [27] Component model, and others.

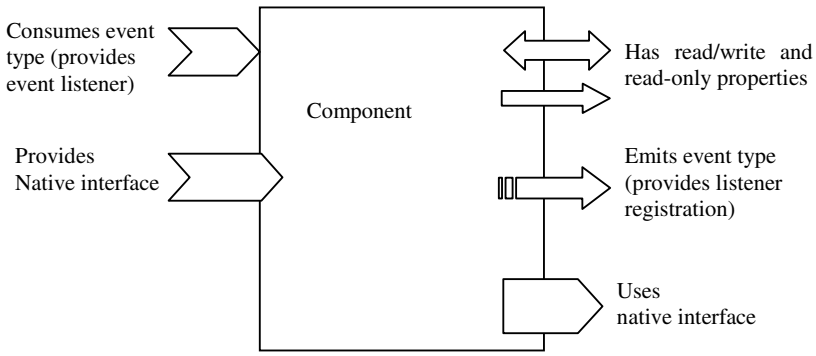

Fig. 3. Abstract Component Model

The model defines the interface provided by a component in terms of three styles of potential interaction:

- The Component may have zero or more properties, i.e. named values, that characterise it. Readable properties expose aspects of the internal state or configuration of the Component, while writeable properties allow the Component to be configured or interacted with. Readable properties are typically active, i.e. they generate events when they are modified by the component, so that they can be monitored by other Components. This corresponds to JavaBean and C# properties.
- The Component may be (a) a producer and/or (b) a consumer of particular kinds of events. Unlike properties, events are ephemeral and a late-joining observer will not have access to already-emitted events. This corresponds to a JavaBean implementing (a) add/removeXListener methods or (b) the XEventListener interface, or to a C# object with (a) an event or (b) an event delegate.
- The Component may (a) implement ("provide") a certain API, i.e. a set of methods, or (b) be able to use (or require the use of) a certain API as provided by some other component. This corresponds to a JavaBean or C# object (a) implementing a public interface or (b) having a writeable property of the type of that interface.

The Component Container is responsible for mapping between the abstract component model and its realization in a particular language and native component technology.

3.6 Component Interactions

The Component Container is also responsible for all interactions between non-toolkit-aware components. Firstly, the Container must make visible various aspects of its current Components via the data-space. In addition to the Component Advert that we have already seen, the Container will also create Component Property data-items in the data space to reflect the current value of Component properties, and will map component emitted events to data-space events if required.

In addition to this, the Container is also responsible for implementing any connections to the components that it hosts and manages, which are requested via the Locale data-space. The potential interactions in the abstract component model are:

- A writeable property (an input) on one component might be slaved to the value of a readable property (an output) on another component;
- A component which implements a listener for a certain kind of event may have this triggered when another component emits that kind of event;
- A component which makes use of a particular API (interface) may be given a (possibly remote) reference to another component which implements that API so that it can invoke methods on it.

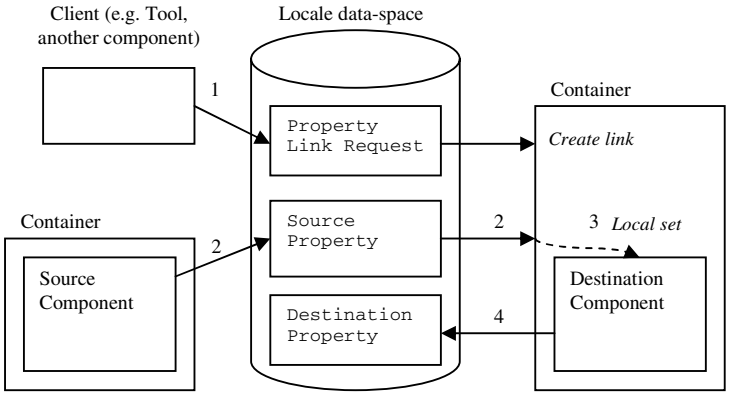

Fig. 4. Linking Component Properties

For example, the linking of a writeable (input) and a readable (output) property is illustrated in figure 4. The Property Link Request data-item requests that a named property on a certain component be linked to another named property on another (or the same) component. For as long as such a request exists the Container which manages the destination Component will (2) monitor the data-space to track the value of the source Component Property and (3) apply this value to the local (destination) Component Property, which it turn (4) will be visible in the data-space.

As with Component Requests, this stateful data-space idiom allows easy inspection of current system state, both of components and interconnections, and provides direct support for expressing the life-time of interconnections and for recovery from transient and partial failures.

3.7 Model Dynamics

The model shares the current state of the Locale and its components, properties and interconnections through data-items in the data-space. In general it is possible for hosts, Containers, Capabilities, Components, Properties and Interconnections to be added and removed during the execution and running evolution of the application or experience. The actual logic for responding to changes and performing re-configurations is assumed to be encapsulated in a subset of the current components, which themselves are subject to management within the same framework. In many cases, such components will also require direct access to the toolkit APIs, in order to monitor aspects of the Locale which are not directly available via their current explicit interconnections.

4 Implementation

The current toolkit implementation has two Container implementations, one in Java which hosts JavaBeans as Components, and one in C# which hosts instances of Common Language Runtime (CLR) classes as Components. Each Container provides mappings to and from the common abstract model described above. At present the Java Container supports dynamic Component life-cycle management through Capabilities and Component Requests, and supports interconnection through readable and writeable Properties. The C# Container currently also supports interconnection through readable and writeable Properties. Components in both Containers can interact through a common Java Locale Master and data-space.

As well as the programmatic APIs, end-user and developer tools are also required to make these environments and applications more readily configured and managed. In particular, we wish to support the construction and management of applications by non-programmers such as artists, designers and curators.

The current toolkit provides two graphical interfaces to the state of the Locale: a generic browser and editor, and an abstract activity display. The generic browser/editor joins and inspects a running Locale and uses the information in the Locale data-space to allow the user to:

- View the currently advertised Capabilities (figure 5(a)) and create a Request for a Component corresponding to one of those Capabilities;
- View the currently active Components, and their current Properties and property values (figure 5(b));
- View the current Property Link Requests and a simple connectivity graph showing Components and their current interconnections (figure 5(c)); and
- Create a Property Link Requests between any two components' properties.

The abstract activity display provides an example of a more impressionistic or ambient display of Locale activity, in which current Components, Properties and interconnection activity are visualized as a changing field of abstract shapes (figure 5(d)).

5 Sample Applications

This section walks through the construction and operation of two simple applications using the current version of the toolkit and some of the currently available components. The applications as described are complete and operational in the current implementation. The test applications took their inspiration from arrangements of ubiquitous devices evident within our existing experiences. However, in contrast to these existing experiences where these arrangements took weeks and even months of coding, the assembly of components and services described in this section took minutes or hours.

The machines and devices employed in the applications are illustrated in figures 5 and 6. Host A is running the Locale Master process and data-space, plus a Java Container which is capable of hosting a camera component (which publishes images grabbed from an attached USB camera as still-image URLs), and a SmartIT component (which exposes the sensor values being returned from directly and indirectly attached SmartIT devices). Host B is running a C# Container which is preconfigured to run a Phidget Host Manager Component; this uses the Phidget COM API to dynamically create Proxy Components for all currently attached Phidgets, in this case an RFID reader and an interface board (with a single analogue slider attached). Host C is running another Java Container, including the Capability to create a media viewer component, which can display various forms of content including images, movies, documents, and presentations. The Tool/Browser machine runs the generic browser/editor application.

5.1 A Remote Controlled Camera

The first example application is a remote controlled camera (see figure 5). Using the browser/editor running on a wireless laptop the user views the advertised Capabilities (figure 5(a)) and sees that Host A advertises (amongst other things) SmartIT and Camera Capabilities. The user issues requests for each of these on Host A. There is also a media viewer Capability on Host C, which the user also requests. The corresponding Containers instantiate the SmartIT, camera and media viewer Components. The SmartIT component uses Host A's serial port to communicate with the attached SmartIT, and via that to communicate with other nearby SmartITs (SmartITs have their own simple peer-to-peer radio protocol). The camera component uses standard OS facilities to interface to the local USB camera.

The user now switches to the running Component view (figure 5(b)), and can inspect the current Properties of these three components. The SmartIT has a number of read-only (output) properties corresponding to the various sensors on the wireless SmartIT; the camera has one read-write (input) property, "CaptureState", which causes it to take a grab a picture, and one read-only (output) property, "url", which it updates each time it grabs an image with a new dynamically generated HTTP URL from which that image can be obtained (the Container incorporates a simple HTTP server). The media viewer component has two input properties and two output proper-

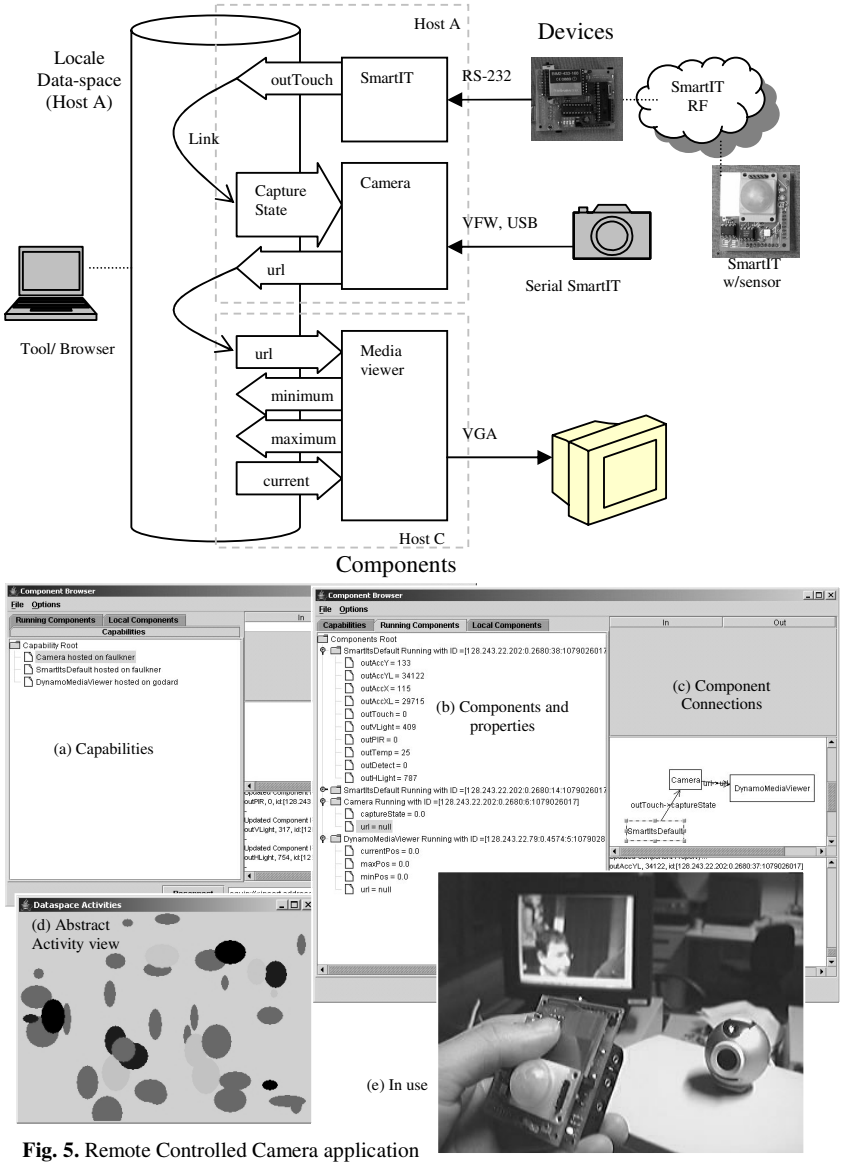

Fig. 5. Remote Controlled Camera application

ties: “url”, the URL of the current content to be displayed, “CurrentPosition”, the viewers current position within the current content (e.g. slide number or movie time), “MinimumPosition”, the start position of the current content, and “MaximumPosition”, the end position of the current content.

The user double clicks on (for example) the “OutTouch” output property of the SmartIT component, and is presented with a dialogue that allows them to connect this

to another property; they choose the "CaptureState" input property of the camera Component. In the same way they connect the camera "url" output property to the media viewer "url" input property.

In the physical environment, the user now touches the touch sensor on the wireless SmartIT (figure 5(e)). The SmartIT regularly broadcasts its current state, and this is received by the other SmartIT connected to Host A's serial port and passed on to the SmartIT component. The SmartIT component (a normal Java Bean) changes its "outTouch" property from 0 to 1, and the property change event causes the corresponding Property data-item to change value from 0 to 1. The camera component Container has been monitoring this property because of the Property Link Request from it to the camera "CaptureState" property, and now sets the camera Component property accordingly. The "CaptureState" setter method of the camera component responds by grabbing an image from the camera and making it available from that Container's built-in web server; the new URL is used to update the camera "url" property. The Container observes this change and updates the corresponding Property data-item in the Locale data-space. In the same way, this change is observed by the media viewer's Container, which sets the media viewer's "url" property accordingly. The media viewer now downloads the image and displays in on Host C's display.

5.2 A Tangible Media Viewer

A second example application creates a simple tangible media viewer application. The user constructs it using the toolkit browser/editor as above. The user has a media viewer component running as in the last example. From the capabilities advertised on Host C they also request a Simple Association Learner component, which learns simple mappings between one input and one output. The user plugs the relevant Phidgets into Host B's USB port(s), and the pre-configured Phidget Host Manager running in the Host C's C# Container creates the corresponding proxy components, in this case an RFID Phidget Component, a Phidget Interface Component and various sensor components including a Phidget Single Sensor Component corresponding to the slider device. From their local Capabilities the user also requests a File Exporter Component, which allows them to publish local files as HTTP URLs.

Inspecting the running component view, the user links together the components as shown in figure 6. The user now places an RFID tag on the Phidget RFID reader, which causes the RFID reader Component's "CurrentTag" property to change. This is propagated to the Learner's "input" property. At this stage the Learner does not know an output for this particular input and outputs null. The user then uses the local GUI of the media exporter to publish a media file (e.g. a Powerpoint presentation) that is currently on their laptop as a URL. This changes the media exporter's "output" property to the newly generated URL, which is propagated to the Learner's "trainingOut" input; this causes the Learner to associate the current "input" (the RFID tag ID) with this preferred output value (the presentation URL). The Learner correspondingly changes its "output" to the new media file URL, and this is propagated to the media viewer which loads and displays the presentation from the laptop.

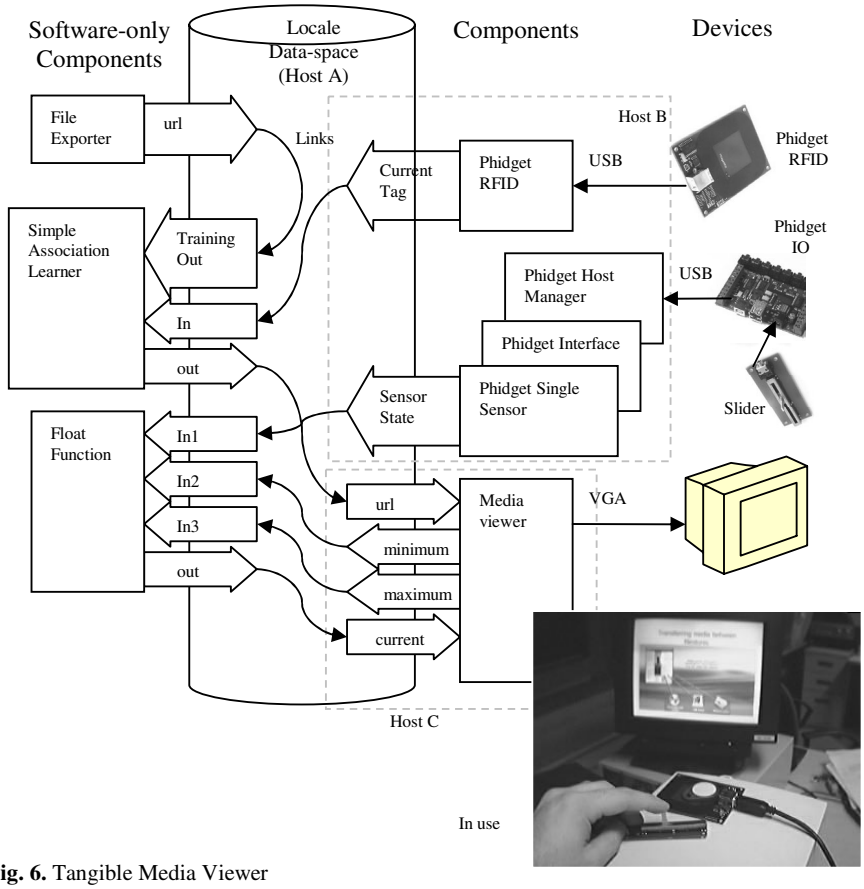

Fig. 6. Tangible Media Viewer

The user may now place other RFID tags on the RFID reader and associate them with other media files from the media exporter. Whenever an RFID tag is placed on the reader, the Learner will output the corresponding learnt URL and the associated file will be displayed by the media viewer.

The user also wishes to navigate within the content using the Phidget slider device. The Phidget Sensor Component's "SensorState" output property varies between 0.0 and 1.0, and user wishes to map this to the full range of the current media clip, as given by the "min" and "max" properties of the media viewer component. The user programs the Float Function Component – a simple script-like component – to map from the slider's range to the full range of the presentation. Now when the user moves the slider device and the position within the media clip changes accordingly.

6 Discussion and Future Work

6.1 Initial Performance and Scalability

The underlying data-space used for communication has a typical end-to-end latency of around 30ms, and a throughput of around 1000 events per second on a typical 1GHz PC. For example, the Tangible Media Viewer application exhibits a latency for navigation within the media clip between moving the slider and repositioning the media of around 50ms. This level of performance is quite adequate for current developments and small to medium scale Locales. However further work – for example using multiple data-spaces – would be required to scale the system to 100s or 1000s of hosts and devices per Locale.

6.2 Persistence

The current version of the toolkit does not yet address persistence of experiences. Two aspects of persistence are required. First, the data-space itself must be persistent in order to make Component Requests and Property Link Requests persistent. This is being addressed in other work on the data-space platform. Second, any additional Component state must also be made persistent, such as the mappings learnt by the Simple Association Learner. One way in which this might be done would be to expose this internal state through additional properties, which could then be made persistent in the data-space. Alternatively, the Container will need to be extended to provide its own persistence facilities with regard to the Components that it is hosting.

6.3 Physical/Digital Identification and Registration

There is a common problem with maintaining a consistent (user) view between the physical sensors and devices present and their software Component representations. This problem has several aspects.

First, some hardware ports and protocols have no standard support for plug and play, so it is not possible to automatically determine what devices and sensors may actually be present at any moment. This is the case, for example, with the serial port used by the SmartITs.

Second, some hardware devices and sensors have no standard self-identity, and so it is not possible to automatically match a particular reading with a specific hardware device. For example, although the Phidget Interface device has a unique ID, the individual sensor sliders do not, and the particular physical sliders that needs to be manipulated can only be determined by explicitly tracing the cabling between the slider and the numbered connection points on the Interface board or by manipulating a specific slider and watching for a correlated change in the component properties.

Third, the identifiers that are available within the software may have no simple mapping to externally visible characteristics. For example, the Phidget Interface device has a unique ID, but it is only made physically visible if it is written on a label and physically stuck onto the device.

Fourth, the user will often want to identify devices and sensors in other frames of reference, for example "this slider" (the user points in space or touches the slider in question). How does the system determine which slider is "this slider"?

We propose to use the Locale data-space as a common context within which we can not only directly represent and manage the software Components, but within which we can also exchange information *about* those components, such as physically where they might be, what they might look like, and so on. In this way, more sophisticated tools and interfaces could support more flexible and user-oriented mappings between physical artifacts and their computational analogues.

6.4 Extensibility

The toolkit currently allows new Java components to be added to the running system in the form of locally deployed JAR files. However there is currently no standard way of performing and managing this deployment within the running system. For example, ideally device Components should be automatically installed when new devices are introduced to the system (as with Plug and Play). Similarly, users should have simple mechanisms to add new software Capabilities to their Locale, such as new media handling components. For example, this might be done through tokens (such as special management RFID tags) identifying remotely available component downloads, or through mobile storage media such as USB memory devices or memory cards.

6.5 User Interfaces

The current user interfaces are prototypes for development and testing, but they already make apparent some of the contrasting and complementary representations or views that might be required for various kinds of use. The generic browser/editor has distinct views for Capabilities (as a tree), active Components and their Properties (as a tree), current links to and from a selected Component (as a table), and active Components and their interconnections (as a 2D graph layout). The abstract Locale visualization provides a complementary "ambient"-style view of Locale complexity and activity.

For end-user programming and configuration there is also a clear need for much simpler and more accessible interfaces, such as the Jigsaw Piece component editor and Tangible linking tool used in the Accord project [23]. There is also ongoing work at other partner sites considering the design possibilities of other approaches to programming and configuration within a specifically domestic setting.

6.6 Programming, Scripting and Training

The toolkit model makes a relatively strong distinction between the creation of components, and the assembly and configuration of applications or environments using such components. In most cases components will be created by programmers within some software development environment that is outside the scope of the toolkit. However component assembly and configuration should be accessible to users without specific programming skills, with the support of various tools that are part of the toolkit.

The presence of scripting components – of which the Float Function is a simple example – provide a bridge between component development and configuration, allowing relatively light-weight specification of elements of functionality during the deployment process, supported by the toolkit.

Components which learn and/or can be trained – such as the Simple Association Learner – also provide a general and flexible mechanism for tailoring the behavior and responsive characteristics of the application again without resort to programming. This appears to be a particularly helpful and accessible approach for end-users and other non-programmers [24] (as well as being faster and less error prone than explicit programming for some classes of behavior). This is a particular emphasis of the activity which involves various partner sites.

6.7 Dynamic and Federated Environments and Applications

As already noted, this first version of the toolkit presumes a reasonably self-contained and well-defined (all be it time varying) deployment situation, such as a single house or museum. We believe that much of the model and infrastructure will generalize to also support more dynamic and federated deployment settings.

Our first approach to this issue will be to add explicit support for multi-Locale discovery, and for stereotypical interactions between Locales. For example, an autonomous wearable computer which is not exclusively tied to a particular Locale would be become its own Locale. Peer-to-peer-aware components within the wearable computer Locale and – for example – a house Locale would perform mutual discovery and dynamic partial bridging between the respective Locales to provide a framework for dynamically negotiated coordination.

6.8 Toolkit Availability

The toolkit source can be downloaded from a public CVS server, details of which are available at . This includes the run-time, tools and components as presented in this paper.

References

1. Rogers, Y., Scaife, M., Harris, E., Phelps, T., Price, S., Smith, H., Muller, H., Randell, C., Moss, A., Taylor, I., Stanton, D., O'Malley, C., Corke, G. & Gabrielli, S. (2002) 'Things aren't what they seem to be: innovation through technology inspiration'. Proceedings of DIS2002, London, 25-28 June, 373-377.
2. Barry Brown, Ian MacColl, Matthew Chalmers, Areti Galani, Cliff Randell, Anthony Steed (2003) Lessons from the lighthouse: Collaboration in a shared mixed reality system. In: Proceedings of CHI 2003, Ft. Lauderdale, p577-585, ACM Press
3. Crabtree, A., Rodden, T., Hemmings, T. and Benford, S. (2003) "Finding a place for UbiComp in the home", Proceedings of the 5th Annual Conference on Ubiquitous Computing, October 12th 15th, Seattle:Springer.
4. Weal, Mark J., Michaelides, Danius T., Thompson, Mark K., and De Roure, David C. (2003) 'The Ambient Wood Journals - Replaying the Experience'. Proceedings of ACM Hypertext'03, The fourteenth conference on Hypertext and Hypermedia 2003, Nottingham, UK.
5. Flintham, M, Anastasi, R, Benford, S D, Hemmings, T, Crabtree, A, Greenhalgh, C M, Rodden, T A, Tandavanitj, N, Adams, M, Row-Farr, J (2003), Where on-line meets on-the-streets: experiences with mobile mixed reality games in CHI 2003 Conference on Human Factors in Computing Systems ACM Press Florida, 5-10 April 2003.
6. Brown, B., and M. Chalmers (2003) Tourism and mobile technology. In: K. Kuutti, E. H. Karsten et al (Eds.), ECSCW 2003: Proceedings of the eigth european conference on computer supported cooperative work, Helsinki, Finland, p335-355, Dordrecht: Klewer Academic Press.
7. Myers, B., User interface software tools**,** ACM Transactions on Computer-Human Interaction (TOCHI), Vol 2, No 1, pp 64 - 103, 1995
8. Roseman, M., Greenberg, S., Building real-time groupware with GroupKit, a groupware toolkit**,** ACM Transactions on Computer-Human Interaction (TOCHI), Vol 3, No 1, pp 66-106, 1996.
9. Myers, B., Hudson, S., E., Pausch , R., Past, present, and future of user interface software tools **,** ACM Transactions on Computer-Human Interaction (TOCHI), Vol 7, No 1, pp 3-28,2000]
10. Bill Segall, David Arnold, Julian Boot, Michael Henderson and Ted Phelps, "Content Based Routing with Elvin4," Proceedings AUUG2K, Canberra, Australia, June 2000.
11. The Context Toolkit: Aiding the Development of Context-Enabled Applications, Daniel Salber, Anind K. Dey and Gregory D. Abowd. In the Proceedings of the 1999 Conference on Human Factors in Computing Systems (CHI '99), Pittsburgh, PA, May 15-20, 1999. pp. 434-441.
12. Greenberg, S. and Fitchett, C. (2001) Phidgets: Easy Development of Physical Interfaces through Physical Widgets. Proceedings of the UIST 2001 14th Annual ACM Symposium on User Interface Software and Technology, November 11-14, Orlando, Florida, p209-218, ACM Press.
13. Phidgets Inc., Phidgets. http://www.phidgets.com (verified 2004-03-12)
14. L.E. Holmquist, H. Gellersen, G. Kortuem et al. Building Intelligent Environments with Smart-Its. IEEE Computer Graphics and Applications, special issue on Emerging Technologies, January-March 2004, pp. 56-64.
15. System architecture directions for network sensors, Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, Kristofer Pister . ASPLOS 2000, Cambridge, November 2000
16. Ballagas, Ringel, Stone, Borchers. iStuff: A Physical User Interface Toolkit for Ubiquitous Computing Environments Proceedings of CHI 2003, p. 537-544

17. Brad Johanson, Armando Fox, The Event Heap: A Coordination Infrastructure for Interactive Workspaces, Proceedings of the Fourth IEEE Workshop on Mobile Computing Systems and Applications, 83-93, ISBN:0-7695-1647-5, IEEE Computer Society, Washington, DC.
18. The Interactive Workspaces Project: Experiences with Ubiquitous Computing Rooms. Brad Johanson, Armando Fox, Terry Winograd. IEEE Pervasive Computing Magazine 1(2), April-June 2002.
19. Challenge: Recombinant Computing and the Speakeasy Approach. W. Keith Edwards, Mark W. Newman, Jana Z. Sedivy, Trevor F Smith, and Shahram Izadi. Proceedings of the Eighth ACM International Conference on Mobile Computing and Networking (MobiCom 2002). Atlanta, GA. September 23-28, 2002.
20. Macromedia, Macromedia Director, http://www.macromedia.com/software/director/ (verified 2004-03-12).
21. Cypher, A. EAGER: Programming repetitive tasks by example. In Proceedings of CHI' 91. ACM Press
22. Sohn, T., Dey, A.K.. iCAP: An Informal Tool for Interactive Prototyping of Context-Aware Applications. Interactive Poster in the Extended Abstracts of CHI 2003, ACM Conference on Human Factors in Computing Systems, pp. 974-975, April 5-10, 2003.
23. Humble, J., Crabtree, A., Hemmings, T., Åkesson, K-P., Koleva, B., Rodden, T., Hansson, P., "Playing with the Bits - User-configuration of Ubiquitous Domestic Environments", Proceedings of the Fifth Annual Conference on Ubiquitous Computing, UbiComp2003, Seattle, Washington, USA, 12-15 October 2003
24. Dey, Anind K., Hamid, Raffay, Beckmann, Chris, Li, Ian, and Hsu, Daniel. *a CAPpella*: Programming by Demonstration of Context-Aware Applications. To appear in the proceedings of CHI 2004
25. Sun, Jini Technology Core Platform Specification, http://wwws.sun.com/software/jini/specs/core2_0.pdf (verified 2004-03-12)
26. Sun Microsystems, JavaBeans, http://java.sun.com/products/javabeans/ (verified 2004-03-12)
27. Erik Meijer, John Gough, Technical Overview of the Common Language Runtime (2000) http://research.microsoft.com/~emeijer/Papers/CLR.pdf (verified 2004-03-12)
28. Mark W. Newman, Jana Z. Sedivy, Christine Neuwirth, W. Keith Edwards, Jason I. Hong, Shahram Izadi, Karen Marcelo, Trevor F. Smith, Jana Sedivy, Mark Newman: Designing for serendipity: supporting end-user configuration of ubiquitous computing environments. Symposium on Designing Interactive Systems 2002: 147-156.
29. Universal Description, Discovery, and Integration Consortium (2004). UDDI Version 3 Specification, March 2004. http://uddi.org/pubs/uddi-v3.00-published-20020719.htm
30. Christensen, E., Curbera, F., Meredith, G., and Weerawarana, S. (2001). Web Services Description Language (WSDL) Version 1.1. W3C Note, March 2001. http://www.w3.org/TR/wsdl
31. Gudgin, M., Hadley, M., Mendelsohn, N., Moreau, J-J., Nielsen, H.F. (2003). SOAP Version 1.2. W3C Recommendation, June 2003. http://www.w3.org/TR/soap12
32. Kindberg T., & Barton. J., A Web-Based Nomadic Computing System. In Computer Networks, Elsevier, vol 35, no. 4, March 2001, pp. 443-456.