BIT – A Framework and Architecture for Providing Digital Services for Physical Products

Christof Roduner
ETH Zurich
Switzerland
http://people.inf.ethz.ch/rodunerc/

Marc Langheinrich
University of Lugano
Switzerland
http://www.inf.usi.ch/langheinrich/

***Abstract*—Mobile phones are increasingly able to read auto-id labels, such as barcodes or RFID tags. As virtually all consumer products sold today are equipped with such a label, this opens the possibility for a wide range of novel digital services building on physical products. In this paper, we discuss the problems that arise when such novel applications are deployed, and present a unified system architecture for providing mobile phone-based digital services in the *Internet of Things*, called BIT. BIT aims to be a "single point of interaction" for users when accessing the services of a variety of tagged objects. BIT also aids service developers and product manufacturers in deploying services linked to tagged products, by providing a cross-device development and deployment framework. We have used BIT to quickly implement nine diverse services in a prototypical fashion, and report on our inital experiences with the framework.**

I. Introduction

Personal mobile devices, such as mobile phones and PDAs, represent an important building block in many of the systems envisioned for the Internet of Things. By using so-called *auto-id tags*, i.e., visual markers or labels based on *Radio Frequency Identification* (RFID) technology, mobile devices can quickly and efficiently identify all sorts of physical objects. Most mobile phones already feature an integrated camera that is often capable of decoding the omnipresent barcodes that can be found on virtually all consumer goods [1]. On top of this, some mobile phones are already capable of reading passive RFID tags, using a *Near Field Communication* (NFC) module that is either built-in or can easily be attached to the phone.[1]

From an economic perspective, the convergence of auto-id technology and mobile phones opens many very attractive opportunities for businesses. While the benefits of auto-id tags were earlier limited to internal business processes (e.g., enhanced efficiency in supply chain management), it is now possible to leverage this technology throughout a product's life cycle. One of the most promising prospects is that businesses can establish a direct link to consumers through simple phone-mediated interaction with the physical product:

- *Manufacturers* can deliver added value to customers by enriching their physical products with digital services that can be accessed in a straightforward and intuitive way. Unlike the physical product itself, such digital services are not static and can be evolved over time as well as personalized, thereby ensuring an ongoing appeal and repeated interaction with the product.
- *Third-party businesses* can equally offer independent product-related digital services. A consumer advocacy organizations, e.g., can "link" a review directly to a physical product, thus making it easily accessible when a buying decision is made in a brick and mortar store.
- *Consumers* in turn can retrieve information and services directly from a physical product, without the intermediate step of a manual search on the web. Additionally, data that allows the user's context to be inferred (e.g., location) can also be taken into account.

Typical usage scenarios of mobile phone-based services for tagged products include, e.g.: a system warning users with allergies that a certain product contains allergenic ingredients [1]; a price comparison service informing shoppers that a product at hand costs less at a nearby store;[2] a product review system that allows users to read product-related user feedback, third-party information (e.g., carbon footprint, fair trade seals), as well as rate items in-place [3]; a system to re-order consumables (such as printer cartridges or coffee capsules) by simply scanning a device (such as a printer or coffee maker); accessing the manual or receiving diagnostic information for a faulty appliance on one's mobile phone; and controlling a device through an extended UI on the phone via an embedded NFC interface [4].

While some of the above scenarios are nothing more than a vision as of now, several of them have in fact already been publicly fielded (e.g., ShopSavvy, Google Shopper). However, existing systems are typically one-shot, proprietary developments that repeatedly re-implement largely identical functionality (i.e., auto-id-based information lookup/feedback). The BIT framework presented here aims at offering a unified architecture that not only simplifies application development and deployment, but also offers consumers a "single point of interaction" for all available services relating to auto-id objects – a dedicated *Browser for the Internet of Things (BIT)*.

When a new physical object is detected, BIT transparently downloads, installs and starts the software needed to use its digital services. It allows users to interact with their environ-

[1] See the Nokia 6131 NFC, or the sleeve solution for the iPhone planned by Visa [2], respectively.

[2] E.g., ShopSavvy at www.biggu.com/apps/shopsavvy-android/ or Google Shopper, both for Android phones.

ment spontaneously, as they no longer need to manually find and install the corresponding application. When an object is detected, BIT also invokes the various applications provided by different parties that may offer a service for the given product. The available services are presented in an overview, freeing users from the need to try one application after another in order to gain a comprehensive picture. For developers, BIT facilitates the creation of such services by providing a number of convenient abstractions, such as an implementation-agnostic communication interface masking whether an object is equipped with a barcode, NFC, or Bluetooth.

II. Related Work

The idea of a personal and portable device that would allow its users to retrieve background information on consumer products has been discussed for a while. An early example is the "personal shopping assistant" [5]. It was proposed in 1994 as a mobile device of the "size of a walkman" with an integrated barcode scanner that could be carried around a retail store by shoppers. It would show product information and also keep a running total of all items purchased. In the last 10 years, a number of such "m-commerce" systems have explored the potential value that mobile devices can offer for applications around consumer products, e.g., [6]–[9]. None of these projects have investigated how the many application ideas can be integrated in a consistent, unified framework. Neither have they looked into the specific needs that arise when all these applications are to run on the end-user's own mobile device that cannot be assumed to be available for exclusive use by these services.

Portable shopping assistants are a specific instance of *context-based information access* – in their case the context is a tagged object. One of the most well-known projects in this area is probably Hewlett-Packard Labs' Cooltown [10], which provided a blueprint for many later projects by others. Cooltown proposed an infrastructure based on world wide web technologies to create a "web presence" for people, places, and things. These entities had their own web pages, where they were able to offer information and services. Just like BIT, Cooltown envisioned a universal interaction device to be used with any object and appliance that a user may encounter. The most notable difference to our work is that Cooltown relied on web pages to deliver information. Firstly, this complicates the use of multiple information providers, as their information would be spread among individual pages.[3] Also, the use of a web-based interface to appliance control has significant security drawbacks. If an appliance, such as a coffee maker, is accessible via the web, some form of authentication is needed in order to prevent unauthorized access. BIT in turn can leverage short-range communication technologies such as NFC and Bluetooth that naturally restrict access to users who are in the appliance's physical proximity.

A number of projects have explored the use of mobile phones in order to interact with and control smart environments. The Physical Mobile Interaction Framework (PMIF) [12] provides developers with a generic framework to write applications that support different interaction techniques in smart environments, such as touching, pointing, and scanning. The PERCI (PERvasive ServiCe Interaction) project [13] also aims at facilitating physical mobile interaction, however, the focus lies on the automatic generation of user interfaces from service descriptions based on Semantic Web Services. The REACHeS (Remotely Enabling and Controlling Heterogeneous Services) project [14], finally, proposes a system that implements the universal remote control paradigm. NFC tags are used to enable users to physically interact with their environment. While all of these projects are related to our work, our focus is more on individual, low-value, physical products than on smart environments. We aim at providing a runtime environment in which services offered by a large number of interested parties can be deployed and executed. In contrast to many other scenarios discussed in the community, the physical objects tend to be more inexpensive in our examples, and both tagged products as well as interaction devices are highly mobile. The emphasis of our research is thus on a lightweight approach that allows for the fast and easy creation of new services in this specific domain.

III. Technical Challenges

While the scenarios outlined in section I above are likely to be appealing to most end users, developers of such services face a number of challenges today, in particular regarding cross-platform development, hardware support, and user interaction.

Even though Google's Android and Apple's iPhone platform have hugely popularized mobile application development, moving beyond these two platforms is still a cumbersome process. Firstly, today's fragmented mobile phone market means that service providers must potentially write a separate version of the same application for half a dozen platforms or more in order to cover a majority of devices in use. Secondly, toolchain and framework support, especially on legacy platforms, is often still poor. Consequently, product manufacturers with an interest in providing digital services for their products might not have the required expertise in software development to do so. Creating an in-house team to do mobile application development, or even outsourcing it to a specialized development company, is costly, and for many of the applications described above may hardly be justified. While the same applies, to some extent, to traditional websites accompanying a physical product, the programming of an interactive website is still a considerably easier task than mobile application development, requiring skills that are much easier to acquire.

An obvious answer to this dilemma, which might also solve cross-platform development issues, might be the use of a web-based solution that runs in the pre-installed phone browser. The problem with this approach is that a mix of HTML, JavaScript, and server-side applications per se does not enable developers to access phone-specific hardware, such

[3] While the need to aggregate resolution services is mentioned in [11], it is not detailed how this could be done.

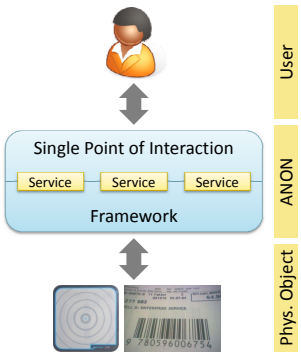

Fig. 1. *BIT overview.* BIT addresses two distinct needs: It represents a "single point of interaction" for users, freeing them from having to try individual applications one after another, and providing a software framework for developers that simplifies the deployment of services for tagged objects.

as barcode or RFID readers, as well as GPS receivers. Another problem arises from the still relatively high latency in today's mobile networks. Applications based on web technology require frequent request-response cycles that are triggered by simple user input. Given the latencies commonly observed, this has a negative impact on user experience. Finally, there are still some places without network coverage, such as subway stations or even large supermarkets. Even without connectivity, a physical object may offer information or services directly to the mobile phone through, e.g., NFC technology or Bluetooth, which could not be supported in a web-based solution.

From a usability perspective, another challenge stems from the necessity to integrate the various services into a single interaction experience. If, as today, each service is deployed as a separate application, this results in the user's phone being littered with a large number of small applications. Worse yet, the execution of these applications is not automatically coordinated. In order to use a service, the user first needs to manually start the corresponding application before a product can be scanned. Since objects might not be associated with a given service, scanning will not render a result in these instances, quickly leading to user frustration. A user's need to simply see "everything my favorite services offer for this product" cannot be addressed in today's fragmented application environment.

We believe that the majority of these challenges can be best addressed by a unified framework and architecture that offers a single entry point into the *Internet of Things*. Such an architecture (see Figure 1) can offer a single runtime environment (i.e., a single application) that is home to all sorts of services as described above. From a user's point of view, this application represents a one-stop shop for interaction with auto-id tagged physical products – a dedicated browser application that allows one to explore physical products and their services. From a developer's point of view, the architecture provides a software framework that facilitates the creation of product-related digital services.

IV. Requirements

Based on the above considerations, we can derive a number of requirements that our BIT system needs to fulfill in order to support the wide array of services that can be implemented for tagged products.

- *Discovery and presentation.* When a tag is read, BIT must discover which services are available and present this information to the user. Optionally, BIT must be able to retrieve status information from an appliance (e.g., an error code indicating a paper jam in a printer).
- *Lifecycle management.* BIT needs to be able to dynamically obtain, execute, cache, and update the code needed to offer a particular service on the mobile phone.
- *Unified user experience.* BIT must coordinate the execution and presentation of all the services that might get triggered by a single barcode scan.
- *Platform and device independence.* A service should run on any platform that provides an BIT implementation.
- *Programming abstractions.* BIT should free developers from dealing with, e.g., a particular tagging technology (e.g., RFID, barcodes) or communication technology (e.g., Bluetooth). Additional contextual information, such as the user's location, should also be accessible.
- *Infrastructure integration.* Most product-related information and services will be stored in a backend infrastructure. In order to discover and retrieve this data, BIT must seamlessly integrate with those infrastructures.
- *Storage.* Certain services, such as an allergy checker, need to permanently store data (e.g., the user's food allergens). BIT must provide persistent storage for such services.
- *Security and privacy.* BIT must ensure that a service can only access those resources for which it is authorized. As BIT mediates all interactions between a user and *every* physical product's digital offerings, it must also minimize the risk of unauthorized tracking.

V. Architecture

A. Framework Core Concepts

The BIT framework is built around a number of concepts, which will be presented in this section.

Perspectives. BIT's user interface is divided into two major parts: the *aggregation perspective* and the *exclusive perspective*. In response to scanning a tagged product, the aggregation perspective provides an overview of information and services that are available. As its name suggests, this perspective aggregates data from various sources in a single view. Figure 2 shows an example of BIT's aggregation perspective. As soon as a user selects an item in the aggregation perspective, the corresponding service is opened in the exclusive perspective for a more detailed view. In this perspective, the service can exclusively use the full assigned screen area to display more detailed information (e.g., directions to another store selling the same product for less). Unlike in the aggregation perspective, an applet can also collect user input.

Fig. 2. Browser aggregation perspective showing overview of information and services available for a tagged object.

Applets. Services and information regarding a physical product are provided through *applets*, i.e., small programs that are executed within BIT. The functionality offered by an applet can range from the very simple task of displaying some information on a physical object to more complex procedures, such as gathering user input and communicating with a physical appliance.

Runlists. Applets can be part of a *runlist*, which groups a number of applets that a user considers relevant in a certain everyday context. For example, while users may be interested in product reviews during a shopping trip, they may prefer not to see such information while at home. When a tagged object is detected, BIT first invokes all applets that form the runlist, passing the tag read to every applet. The applet is expected to generate a terse output (e.g., "Caution: contains peanuts") that can be empty optionally. All non-empty output is collected by BIT and displayed in the aggregation perspective as shown in Figure 2. The example in said figure has been generated by a runlist with an "Allergy Checker" applet placed at its top. By assigning this applet a prominent position in the runlist, the user ensures to immediately grasp the information that is most important to her or him. If an applet decides not to return any output (e.g., no price information is available for a price comparison service), it is omitted from the results list in order to unclutter the interface.

Ranged vs. singular applets. While some applets can be used in connection with any tagged object, others can provide a meaningful service only in connection with a specific one. An example of the first kind is an applet that offers product reviews. It can show reviews for a potentially large number of products and should be invoked whenever the user scans a tagged object. This is ensured by including the applet in a runlist, which requires it to be individually downloaded and installed in the browser for frequent use. Such applets are called *ranged*.

An example of the second kind is an applet to control an appliance. This sort of applet can provide a service only for a specific tagged object, such as a particular model of a coffee maker. Including it in a runlist makes little sense, since it cannot react in a meaningful way in response to the recognition of any other object. In contrast to ranged applets, so-called *singular* applets should start automatically as soon as the user interacts with the corresponding tagged object. However, in order to avoid the risk of unsolicited applets popping up as soon as an object is scanned, we only allow the manufacturer to provide a singular applet for an object. In addition, singular applets cannot persistently store data in order to have them available during a subsequent execution (unlike ranged applets). After a singular applet is stopped, it is removed entirely from the browser without leaving any traces behind (though caching of applet code is possible). This restriction represents an additional safeguard to allow for spontaneous, possibly one-off interaction with services whose source cannot necessarily be trusted.

Virtual reads. Similar to a web browser, BIT allows users to bookmark tagged objects. It also keeps a history of all tagged objects that were scanned. When a bookmark or history item is selected, BIT internally creates a virtual read, which is processed as if the physical object had been recognized. This allows for the display of up-to-date information in bookmarks (e.g., latest prices in a price comparison applet).

Open lookup infrastructure (OLI). BIT leverages our previously developed open lookup infrastructure (OLI) [4]. OLI allows both manufacturers as well as third parties to offer applets and further information for any tagged object. It also provides a context-aware lookup mechanism, which helps users to find those services that are most relevant to their respective situation. While BIT relies on OLI to discover applets and metadata for tagged objects (see below), any similar infrastructure (such as the Object Naming Service [15]) could be used instead.

Communication endpoints. BIT provides applet developers with the abstraction of *communication endpoints*. An endpoint represents a logical connection to a physical object that offers bidirectional communication through Bluetooth, TCP/IP, HTTP, NFC, or a similar technology. It allows developers to write applets that can communicate with physical objects without needing to worry about the concrete underlying communication technologies. When an object is recognized, the browser detects whether it has bidirectional communication capabilities. It does so by inspecting its metadata, which was made available by its manufacturer via OLI. If the object is found to offer bidirectional communication, an endpoint is created and passed to the applet. As soon as the applet decides to send or receive a message to or from the endpoint, the browser automatically establishes a physical connection to the device through one of the communication technologies that were discovered when the physical object was originally recognized.

B. Handling of Tagged Object Reads

In this section, we will present in detail how BIT proceeds when the user scans a tagged object. The exact steps taken

by the browser in this phase depend on whether the tagged object was recognized while the aggregation perspective or the exclusive perspective was active.

In the **aggregation perspective**, BIT executes the currently active runlist in the background and updates the aggregation perspective accordingly. If no runlist is active or none is defined, all installed ranged applets are executed.

At the same time, BIT checks via OLI whether the tagged object's manufacturer provides an applet for it.[4] If an applet is found, BIT downloads it and – if it is a singular applet – executes it immediately in the exclusive perspective. Note that a manufacturer applet might also be a ranged applet, e.g., the developer of a comparison shopper applet may print a paper flyer touting its features, and include a barcode link for downloading it. In such cases, BIT downloads and installs the ranged applet, but does not execute it right away. Instead, the user can decide whether or not to add it to a runlist.

When a tagged object is recognized while the **exclusive perspective** is active, BIT simply passes the tag read to the applet running in the exclusive perspective. At the same time, BIT again executes the runlist and updates the aggregation perspective in the background.[5]

C. Service Development

We will now present the tools that BIT provides for developers to implement services. The major building blocks are: the **BIT markup language** (BITML) for the definition of services' user interfaces; a **scripting language** for the implementation of the dynamic behavior of services; and the **BIT API**, which allows services to access the functionality provided by the browser.

1) BIT markup language (BITML): To ensure the portability of services across platforms, BIT relies on its own XML-based user interface description. It defines a number of commonly used GUI widgets, whose presentation can be customized using CSS [16]. A plethora of user interface markup languages exist, such as UIML [17], XIML [18], XUL [19], or XAML [20]. BITML shares many similarities with such languages, yet it is much simpler (and therefor less powerful). While BITML could be easily replaced with another language, such as XUL, we opted for creating our own leightweight markup language in order to simplify the development of BIT.

In BITML, an applet's user interface is structured into *views*, which occupy the entire screen estate that is available to the browser's exclusive perspective. Views can have a title, a body, and a menu. This simple structure worked well with the phone hardware of our prototype (see section VI below), though it might work less well on modern touch screen-based devices. However, as the focus of our work was not on portable user interfaces, we decided to stick with our simple UI model for our first prototype.

Listing 1. Excerpt from *main.lua* file.

```
REPOS_URL = "http://repository.example.com/"
PROFILE_REVIEW = "review"

function start_exclusive_perspective(tagged_object)
  -- connect to repository in OLI
  local repos = anon.get_repository(REPOS_URL)
  -- fetch reviews (= resources with profile "review")
  review_resources = repos.lookup_resource({tag_id =
    tagged_object.tag_id, profile = PROFILE_REVIEW})
  anon.show_view("overview")
end
```

Listing 2. Corresponding view *overview*.

```
<view title = 'ReviewCentral.com'>
  <list><? reviews = {}
          for i = 1, #review_resources do
            reviews[i] = anon.json_decode(
              review_resources[i].data.value)
            anon.out('<list_item description="Rated ' ..
              reviews[i].rating .. ' out of 5">')
            anon.out(reviews[i].title)
            anon.out('</list_item>')
          end ?>
    <menu><menu_item action="anon.show_view('details',
       {review = reviews[index]})">Show details</menu_item>
    </menu></list>
</view>
```

Fig. 3. Source code examples taken from review applet (see Figure 5(a)).

2) Scripting: For the implementation of the dynamic behavior of services, we leverage the Lua scripting language [21]. The BIT browser application incorporates a Lua interpreter, which supports the local execution of Lua scripts. Like many web development frameworks (e.g., Ruby on Rails, PHP, JSP), BIT allows developers to mix markup and scripting code to create a dynamic user interface, while also supporting standalone scripts.[6]

Technically, a BIT applet is a ZIP file that contains a Lua script named *main.lua*, which must implement a callback function that is invoked by the browser when the applet is started in the aggregation perspective (for ranged applets) or in the exclusive perspective (for transitory applets). The ZIP file can optionally contain additional script files, views, and other resources, such as images.

On top of the main callback function, BIT offers further callback functions that can be implemented by an applet in order to be notified about subsequent read events or its imminent termination. When BIT notifies an applet about a read event, it passes to the callback function three elements: First, the physical object's tag id. Second, context information (e.g., the status code of an appliance as picked up over NFC). Third, an endpoint object that can be used to communicate with the physical object.

3) API: In addition to BITML and Lua scripting, BIT also offers an API that provides a range of functionality, such as user interface manipulation (e.g., switch between views, show dialogs, open URLs, access phone's vibration module),

[4]Note that in this context, the term "manufacturer" has a somewhat broader meaning. It simply refers to the party that commissioned (i.e., reserved) an auto-id identifier for the physical object. In the case of a movie poster, for example, the "manufacturer" would typically be an advertising company, a movie distributor, or a similar entity.

[5]This ensures that the output produced by other applets is available immediately when the user switches from exclusive to aggregation perspective.

[6]Note that, unlike with web frameworks, all markup and scripting code is interpreted and run on the phone.

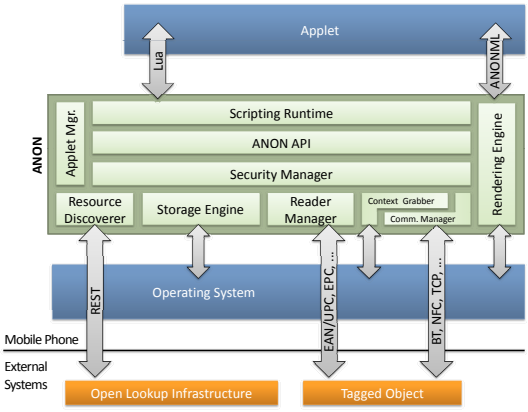

Fig. 4. Architecture of BIT.

persistent storage, and data exchange with various backend services (e.g., access OLI, invoke HTTP requests, serialize and deserialize Lua data structures to and from XML or JSON [22] formats). While space limitations do not allow us to discuss an example in detail, we reproduce some excerpts from the product review applet's source code in Figure 3.

VI. Prototype

Based on the concepts outlined above, we developed a prototype implementation of BIT in the form of a stand-alone browser application. In this section, we will present the high-level architecture of the browser and its main components as well as an overview of the actual implementation of BIT and the technologies used.

A. Components

As a runtime environment for applets, BIT acts as an additional layer between the mobile phone's operating system and the services that are available to users. BIT's role on the mobile phone platform, its interaction with the environment, as well as its main components are illustrated in Figure 4. The functions of these components are as follows:

The **resource discoverer** connects the browser to the open lookup infrastructure. It encapsulates the protocols used to access the OLI and provides a local interface that allows the browser components to find applets. The **applet manager** is responsible for the lifecycle of applets and coordinates their execution. Whenever a tagged object is recognized, the applet manager tries to discover and start applets. It also manages the installation of new applets and keeps track of runlist configurations. It relies on the *reader manager* to obtain notifications on recognized tagged objects, the *storage engine* to load already installed applets, as well as the *scripting runtime* to execute applets. The **scripting runtime** provides the Lua interpreter that is required to execute applets. It also makes sure that the BIT API is available to applets at runtime. The **BIT API** allows applets to use the functionality of resource discoverer, storage engine, reader manager, context grabber, and rendering engine in a standard way that ensures portability of applets across different browser implementations and mobile device platforms. The **security manager** controls an applet's access to the functionality provided through the BIT API. It ensures, e.g., that an applet can only access its own data store. The **rendering engine** is invoked through the BIT API and allows an applet to present its user interface on the screen. It parses the BITML code of an applet's views. The **storage engine** provides means for both (ranged) applets and other browser components to persistently store data in the browser. The **reader manager** is used by the applet manager and, indirectly through the BIT API, applets to recognize tagged objects. It wraps the different tagging technologies that may be available on a particular mobile device (e.g., barcodes, NFC, EPC, etc.) and provides a simple interface that abstracts from the specific labeling standard. The **communication manager** is responsible for establishing and managing bidirectional connections with tagged objects that offer communication capabilities. It is used by the BIT API to provide the endpoint abstraction. The **context grabber** allows applets to acquire information about their context. This includes, for example, location and status information on the last recognized object.

B. Implementation

We implemented a prototype of our browser for the Symbian S60 platform, which is (still) one of the most widespread smartphone platforms. The browser itself is written in Python, which we used because it allows for the rapid development of applications for S60 phones. Python is available on S60 through the PyS60 project [23]. PyS60 allows developers to access *extensions*. An extension is a binary module that is written in the platform's native Symbian C++ language. We used this mechanism to import the BaToo Barcode Recognition Toolkit [1], which offers fast and robust barcode recognition using the phone's built-in camera. The same approach, i.e., a Python extension module written in Symbian C++, was used for the Lua interpreter. We ported the source distribution of the Lua interpreter [21] along with Lunatic Python[7] to Symbian S60. Lunatic Python provides a "bridge" between the Python interpreter and the Lua interpreter. With the resulting S60 module, a PyS60 program, such as BIT, can import and control a Lua interpreter from within Python. The hardware platform that we used was a Nokia E61i mobile phone. While this model does not provide an NFC module, the particular phone we used had been modified by Nokia Research Center [24] and features a built-in UHF RFID reader based on EPCglobal's Class 1 Generation 2 protocol [25].

VII. Example Services and Discussion

In order to validate both our framework design as well as our prototypical browser implementation, we created nine example services for BIT. Our goal was to assess the practical value that our framework can provide for relevant services and to illustrate the range of different service types that are supported. We mostly selected services that had been proposed

[7] http://labix.org/lunatic-python

(a) Customer reviews. (b) Troubleshooting instructions.

Fig. 5. Two example applets implementing a product reviews service (a) and a coffee maker controller service (b).

Applet	Views	V. LOC	S. LOC	Total LOC
Product reviews	2	51	34	85
Political shopping	2	43	30	73
Carbon footprint calc.	1	26	124	150
Allergy checker	2	38	107	145
Price comparison	2	33	69	102
Shopping list	2	38	88	126
Search service	1	24	29	53
Coffe maker controller	13	216	44	260
Self checkout	4	60	98	158

TABLE I
Example applets metrics: NUMBER OF VIEWS, SOURCE LINES OF CODE (LOC) FOR VIEWS, LOC FOR SCRIPT FILES, AND LOC TOTAL.

or taken up and refined by others in the Internet of Things community [1], [26]–[30].

In particular, the services we prototypically implemented are: (a) A *product reviews* applet that allows users to browse other consumers' written opinions and numerical ratings for a product (see Figure 5(a)). (b) A *"political shopping"* applet by an imaginary consumer pressure group. (c) A *carbon footprint calculator*, which lets consumers review the carbon emissions produced by a product. Users can add a product's emissions to their personal record, review their total carbon footprint, and reset their personal record. (d) An *allergy checker* applet, which notifies users when a product should not be consumed according to their dietary requirements. Users can configure these requirements in the applet. (e) A location-aware *price comparison* applet, which displays cheaper nearby offerings of the same product. The applet leverages Google Maps to offer detailed directions when an alternative store is selected. (f) A *shopping list* service, which allows users to prepare a personal shopping list by scanning the products that should be bought on one of the next shopping trips. (g) A *search service*, which allows users to find standard web pages on the internet that are related to the product at hand and that are relevant in the user's current context.[8] (h) A *coffee maker controller*, which allows users to operate the appliance through BIT. The applet relies on Bluetooth for bidirectional communication with a coffee maker, simulated by an application running on a laptop. This application can simulate the status of an actual coffee maker (e.g., "need filter change"). Depending on this status, the applet automatically shows corresponding troubleshooting instructions as soon as it is started (see Figure 5(b)).[9] (i) A *self checkout* applet that allows grocery store patrons to scan goods before they put them in their shopping baskets. Once they have collected all items, users can proceed to a payment point in the store to scan a special "pay" barcode. The endpoint abstraction is used to connect the applet with a cash terminal that accepts the customer's credit card payment.

These nine example applets show how BIT supports the development of a broad range of different services for tagged objects. Table VII shows how many views and how many source lines of code[10] were needed to implement the services as described. While lines of code are certainly not the most accurate metrics (and it is to be expected that a scripting language like Lua requires relatively few lines of code), these numbers should help to give an impression of the complexity of developing a service for the BIT framework.

For illustrative purposes, we also compared the most complex applet, the coffee maker controller, with the "appliance interaction device" (AID) presented by Roduner et al. [31]. Our applet is essentially a re-implementation of their Java ME-based AID and offers the same functionality. On top of this, our applet also retrieves and updates the appliance's settings, while their version does not communicate with the device. Despite its more limited functionality, the Java ME-based implementation required 931 source lines of code[11], compared to the 260 lines used by the BIT applet.

An important benefit from the use of an interpreted scripting language is the platform independence of BIT. However, as applets are distributed in source code form, it is easy for anyone to look into their implementation, borrow ideas, or create derivatives. Appliance manufacturers, e.g., would effectively disclose the interfaces to their physical products by offering a controller applet. While this may be perceived as a threat to their business by some, note that this transparency is not unique to BIT: The bustling Web 2.0 community makes heavy use of technologies that disclose source code, which allows developers to learn from each other and build new, innovative mashups.

As applets contain Lua scripts, they represent mobile code, which poses some risks to its host. BIT mitigates these risks through the use of the Lua interpreter as a sandbox. As a virtual machine that lies above the operating system, the scripting runtime may offer some degree of protection for the host [32]. Access to operating system functionality is possible only through the BIT API and can be controlled by the security manager. Applets themselves are isolated from each other.

[8]This applet relies on the search service available in OLI [4].

[9]Note that BIT makes no attempt to standardize the message exchange between applets and appliances. As applets are downloaded dynamically, developers can easily implement the specific protocol supported by the appliance. BIT supports this process with an API for XML and JSON data handling.

[10]We counted physical lines of code by using the UNIX `wc -l` command.

[11]Christof Roduner, personal communication, December 3, 2009.

Every applet has its own state in the Lua interpreter, which is unavailable during the execution of another applet.

A major privacy concern in BIT is that all permanent applets which are part of an active runlist are notified about every single tagged object that the user scans. This could allow service providers to gain detailed insight into a user's everyday scanning activities — no matter whether a specific service is actually used or not. BIT mitigates this problem by limiting the information an applet can transmit when it runs in aggregation mode. In order to prevent applets from revealing any information that could be used to identify the user, the security manager blocks all communication operations bar resource lookups in OLI. In these lookups, applets can only use a very restricted request syntax that prevents the transmission of personally identifiable information to the backend infrastructure.

Note that these restrictions apply in aggregation mode only. In exclusive mode, an applet can communicate freely because, unlike in aggregation mode, it is only notified about tagged objects that are recognized while the user explicitly uses the service. In order to prevent applets from logging reads during aggregation mode and then later sending them during exclusive mode, the security manager also blocks write operations to the storage manager while the applet runs in aggregation mode. Read operations are still possible and necessary (e.g., for an allergy checker to fetch a user's dietary restrictions).

VIII. Summary and Outlook

Since we believe that the idea of enriching physical products with digital information and services will become increasingly popular, we see the need for simpler ways to build such services. To address this need, we presented a framework and a prototype implementation of BIT – a browser for the Internet of Things if you will – that accommodates such services in the form of lightweight, portable applets. BIT aims to be a "single point of interaction" for users by coordinating the many services that may be linked to a physical object. It frees users from the need to repeatedly scan a product with the many installed applications in order to check which services are available. By dynamically downloading and executing applets, it also enables spontaneous interaction with services that were not previously known to the user. On the other hand, BIT provides a software framework that considerably simplifies the development of mobile phone-based services for tagged objects. We believe that the concepts employed by BIT can be a first step towards our vision of making the development of mobile services for products no more complex than the creation of a traditional product web site.

While our initial prototype was developed for Symbian S60, an implementation for the Android or iPhone platform could take advantage of touch-based interaction. However, as pointed out earlier, the focus of our work was not on user interface concepts. Even with a different user interface, the advantages of a one-stop solution integrating and coordinating the services from different providers remain.

References

[1] R.Adelmann *et al.*, "Toolkit for Bar Code Recognition and Resolving on Camera Phones – Jump Starting the Internet of Things," in *Informatik 2006 MEIS Workshop*, Dresden, Germany, Oct. 2006.

[2] B.Ray, "Leaked release shows Visa plotting NFC iPhone case," *The Register*, May 2010. [Online]. Available: http://www.theregister.co.uk/2010/05/05/iphone_nfc/

[3] S.Karpischek *et al.*, "my2cents – Sharing Comments about Retail Products on Twitter," Poster at Pervasive'10, Helsinki, Finland, 2010.

[4] C.Roduner *et al.*, "Publishing and Discovering Information and Services for Tagged Products," in *Proc. of CAiSE'07*, ser. LNCS. Springer, 2007.

[5] A.Asthana *et al.*, "An Indoor Wireless System for Personalized Shopping Assistance," in *Proc. of WMCSA'94*, Santa Cruz, USA, 1994, pp. 69–74.

[6] R.Bellamy *et al.*, "Designing an E-Grocery Application for a Palm Computer: Usability and Interface Issues," *IEEE Pers. Comm.*, vol. 8, no. 4, pp. 60–64, Aug. 2001.

[7] R.Lawrence *et al.*, "Personalization of Supermarket Product Recommendations," *Data Mining and Knowledge Discovery*, vol. 5, no. 1–2, pp. 11–32, Jan. 2001.

[8] E.Newcomb *et al.*, "Mobile Computing in the Retail Arena," in *Proc. of CHI'03*. Ft. Lauderdale, FL, USA: ACM, Apr. 2003, pp. 337–344.

[9] P.Kourouthanassis *et al.*, "Developing Consumer-Friendly Pervasive Retail Systems," *IEEE Perv. Comp.*, vol. 2, no. 2, pp. 32–39, Apr. 2003.

[10] T.Kindberg *et al.*, "People, Places, Things: Web Presence for the Real World," *Mobile Netw. and Applications*, vol. 7, no. 5, pp. 365–376, 2002.

[11] T.Kindberg, "Implementing Physical Hyperlinks Using Ubiquitous Identifier Resolution," in *Proc. of WWW'02*, Honolulu, HI, USA, 2002.

[12] E.Rukzio *et al.*, "A framework for mobile interactions with the physical world," in *Proc. of WPMC'05*, Sep. 2005.

[13] G.Broll *et al.*, "Supporting Mobile Service Usage through Physical Mobile Interaction," in *Proc. of Percom'07*, White Plains, NY, USA, 2007.

[14] J.Riekki *et al.*, "Universal Remote Control for the Smart World," in *Proc. of UIC'08*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 563–577.

[15] EPCglobal, "EPCglobal Object Name Service (ONS) 1.0.1," May 2008.

[16] B.Bos *et al.*, "Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification," W3C Cand. Rec., Sep. 2009.

[17] M.Abrams *et al.*, "UIML: an appliance-independent XML user interface language," *Comp. Netw.*, vol. 31, no. 11–16, pp. 1695–1708, 1999.

[18] A.Puerta *et al.*, "XIML: A Common Representation for Interaction Data," in *Proc. of IUI'02*. San Francisco, CA, USA: ACM, 2002.

[19] B.Goodger *et al.*, "XML User Interface Language (XUL) 1.0," 2001.

[20] Microsoft, "Extensible Application Markup Language (XAML)," 2008.

[21] R.Ierusalimschy *et al.*, "The Evolution of Lua," in *Proc. of ACM SIGPLAN HOPL III*, San Diego, CA, USA, Jun. 2007, pp. 2.1–2.26.

[22] D.Crockford, "RFC 4627: The application/json Media Type for JavaScript Object Notation (JSON)," Jul. 2006.

[23] J.Scheible *et al.*, *Mobile Python: Rapid Prototyping of Applications on the Mobile Platform*. Chichester, England: Wiley & Sons, 2007.

[24] J. T.Savolainen *et al.*, "EPC UHF RFID Reader: Mobile Phone Integration and Services," in *Proc. of CCNC'09*, Las Vegas, NV, USA, 2009.

[25] EPCglobal, "EPC Radio-Frequency Identity Protocols Class-1 Generation-2 UHF RFID Protocol for Communications at 860 MHz – 960 MHz, Version 1.1.0," Dec. 2005.

[26] F.von Reischach *et al.*, "A Mobile Product Recommendation System Interacting with Tagged Products," in *Proc. of PerCom'09*, Galveston, TX, USA, Mar. 2009, pp. 1–6.

[27] A.Dada *et al.*, "Displaying Dynamic Carbon Footprints of Products on Mobile Phones (Demo)," in *Adj. Proc. of Pervasive'08*. Sydney, Australia: OCG, May 2008.

[28] L.Deng *et al.*, "LiveCompare: Grocery Bargain Hunting Through Participatory Sensing," in *Proc. of HotMobile'09*, Santa Cruz, CA, USA, 2009.

[29] F.Resatsch *et al.*, "Mobile Sales Assistant – NFC for Retailers," in *Proc. of MobileHCI'07*. Singapore: ACM, Sep. 2007, pp. 313–316.

[30] A.Thomas *et al.*, "Grocery shopping: list and non-list usage," *Marketing Intelligence & Planning*, vol. 22, no. 6, pp. 623–635, 2004.

[31] C.Roduner *et al.*, "Operating Appliances with Mobile Phones – Strengths and Limits of a Universal Interaction Device," in *Proc. of Pervasive'07*. Toronto, Canada: Springer, May 2007.

[32] D. M.Chess, "Security Issues in Mobile Code Systems," in *Mobile Agents and Security*, ser. LNCS, G.Vigna, Ed. Berlin Heidelberg New York: Springer, 1998, vol. 1419, pp. 1–14.