Thingies for Dummies
A smart home infrastructure for the rest of us

David Thomas
dtho@itu.dk

Egil Hansen
ekri@itu.dk

ABSTRACT

Author Keywords
TODO.

ACM Classification Keywords
H.5.2 Information interfaces and presentation (e.g., HCI): Miscellaneous.

General Terms
Design, Documentation, Economics, Experimentation, Human Factors, Languages, Management, Measurement, Performance, Reliability, Security, Standardization, Theory, Verification.

INTRODUCTION
More than 20 years after Weiser first defined ubiquitous computing [20] we have yet to see smart homes become commonplace. We believe this is largely due to smart home technology being inaccessible and too complicated for everyday users, and both price, perceived lack of advantages, and challenges in learning new technology scare users away.

In this paper we propose a smart home infrastructure that allow homeowners to cheaply bring new smart home technology into their existing homes in small batches, technology that homeowners will realistically be able to install and manage themselves. Our proposed infrastructure bases itself on hardware that exists in most homes today – Wi-Fi and smart phones – and uses simple and open protocols like HTTP and the WebSocket protocol to provide the communication backbone of the infrastructure. Each individual device, or "Thingy", as we have dubbed them, is supposed to be simple to install and use, but provide value nonetheless. We call this concept *Thingies for Dummies*[1].

Edwards and Grinter describe, in their 2001 paper [8], seven challenges facing the adoption of smart home technology[2], and we believe that through the overarching theme of 'simple' in our proposal – simple infrastructure and simple thingies – we are able to answer the first six challenges, at least partly. Simplifying does imply giving up on the dream of a fully automated intelligent home for the time being, but our proposed infrastructure does not prevent a home from becoming gradually more automated or intelligent as developers become better at leveraging the platform.

When we started doing research for this project we first wanted to see if we could come up with a few use cases where some smart home technology would add so much value, that a normal homeowner would go out and invest time and money to retrofit the home. Through our informal questioning of friends, colleagues, and family, it quickly became clear that the promise of the classic intelligent appliances we see in many office buildings that automatically adjusts their operations through context awareness[3] did not have a strong enough appeal. The only thing that really resonated was various remote control scenarios. Examples included double checking if all the lights are turned off at home, controlling radiators, remotely starting the washer when leaving from work so clean clothes are ready to be hung up to dry when the user gets home, and remotely unlocking the front door to the home if a friend is waiting and the owner is running late. With that in mind, we created the following scenario to guide our design process:

After putting his children to bed, Peter is ready to go to sleep as well. He sits on his bed and picks up his smartphone and navigates to his Home Remote app, where he selects the "Lock down" preset. Once selected, the app sets all the homes actuators into their specified lock down mode, i.e. all doors are locked, lights are turned off, curtains are drawn, and the array of sensors in the house reports back what they see. Peter looks at the status screen and sees a warning indicating an open window in the kitchen. Peter stands up and walks out of the bedroom towards the kitchen. As he walks with his phone in his hand, the management software turns on the light around him and unlocks the doors he is near, while keeping everything else in lock down mode. Once the window in the kitchen is closed, the sensor indicator on his

[1]This is a reference to the *For Dummies* book series that tries to present non-intimidating guides to various topics and that always contains the subtitle "A Reference for the Rest of Us!".

[2]1: *The 'Accidentally' Smart Home*. 2: *Impromptu Interoperability (islands of functionality)*. 3: *No Systems Administrator*. 4: *Designing for Domestic Use*. 5: *Social Implications of Aware Home Technologies*. 6: *Reliability*. 7: *Inference in the Presence of Ambiguity*.

[3]Appliances such as HVAC, light fixtures, window curtains, and automatic doors.

Submitted for evaluation in the Pervasive Computing Lab class at IT University of Copenhagen, Spring 2012.

phones turns green Peter returns to his bedroom for a good night's sleep.

Even if this scenario does not describe a remote control scenario where the user is many kilometers away, it is still remote by our definition. We consider anything remote to be if the user is able to control a thingy without being able to observe how it affects its surroundings. For example, if user A is sitting in room A, and remotely controlling a lamp in room B, and user A is unable to see what is going on in room B, i.e. see the physical lamp turn on and off or see if there are people or other appliances in room B that react to the lamps state change, then it does not matter if room A and room B are right next to each other or on opposite sites of the word, it is a remote controlling scenario because user A has the same information available in both cases.

THINGES FOR DUMMIES INFRASTRUCTURE

The key design decision for *thingies for Dummies* is that it should be simple and inexpensive to get started retrofitting an existing house with thingies and that it should be independent of specific hardware and software platforms. This means not requiring a dedicated, centralized controller, and instead to rely on the existing infrastructure in the home for handling communication and control, i.e. a Wi-Fi network and smart phones. By doing so, we limit the amount of new hardware required to get started, since most homes will already have a Wi-Fi network and at least one smartphone. This lowers the price and reduces the environmental impact. Figure 1 shows an overview of the infrastructure:

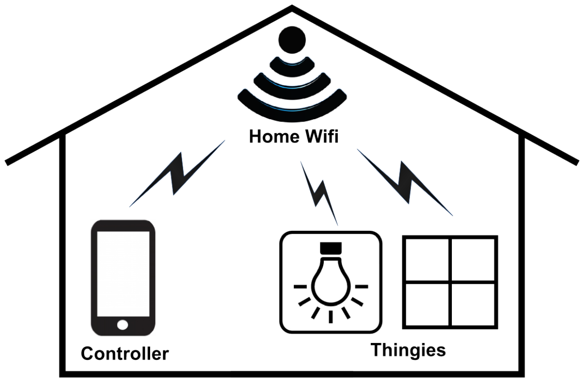


Figure 1. Overview of the infrastructure

An existing Wi-Fi network in the home provides the communications channel. The user's smartphone is connected to this network. A number of thingies – exemplified in the figure as a light switch and a window sensor – are brought into the home and connected to the Wi-Fi network as well. An app on the smartphone then enables the user control and check on the thingies.

In the rest of this section we will describe each of the critical elements in our proposed infrastructure and discuss our design decisions.

Networking, protocols, and Security

With so many networking technologies available care must be taken before picking one above another, however, we arrived at our choice, Wi-Fi, pretty fast. One of our primary requirements was that the network should exist in most homes already. This eliminates quite a few choices right away, leaving us with power lines, which the X10 protocol uses, cellular networks such as 3G, phone line wiring, Bluetooth, and Wi-Fi. Power lines, which has the advantage of reaches all rooms in a home, has problems with the an uncertain network boundaries, since there is usually no special gateway or firewall preventing data from traveling on the power lines to the neighbours power lines, unless extra hardware is added. Power lines are also inherently a wired network, which means an extra bridge device is needed between the power lines and wireless mobile devices if those are to interact with devices connected to it. The phone lines inside the home has similar problems, and is also far less ubiquitous in the home, with most homes only having a few phone plugs. Then there is Bluetooth, which most laptops and phones, even feature phones, come equipped with. Bluetooth is however limited in range. This leaves cellular networks, which are indeed wireless, but also owned and maintained by carrier companies, that charges fees for the data that travels on their network and not all areas are covered equally well. This makes cellular networks the less ideal choice, landing us squarely on Wi-Fi as the preferred option.

There are many advantages to choosing the Wi-Fi as the basis for our infrastructure and a few drawbacks. Users get to reuse their existing Wi-Fi in their home, thus saving money and time setting up a new device. Wi-Fi also adds, assuming it is protected with a passphrase, a virtual boundary to the smart home, a boundary that users understand because they understand that you cannot access the Wi-Fi without the passphrase. Wi-Fi also have the advantage of being a high bandwidth network which enables thingies such as high resolution CCTV cameras.

The biggest drawback related to Wi-Fi is that it is power hungry compared to other technologies such as ZigBee [1]. However, there is a general focus from the Wi-Fi radio chip makers to lower power consumption due to the massive surge of battery driven mobile devices the last few years, so the drawback will matter less as Wi-Fi radios become more power friendly. In addition, for many of the scenarios we have found users to be interested in, thingies will have a permanent power source and in the case where low powered sensor thingies are required, we expect these to use a protocol like ZigBee[1] and be bundled with a special purpose thingy which act as a bridge between the individual ZigBee enabled thingies the Wi-Fi network. The bridge thingy could also perform other tasks so the extra cost would be negligible.

We also discussed the problems with oversaturation and general instabilities of the average home Wi-Fi network. This is of course very dependent on the model of the wireless access point used. Better models with good Wi-Fi radios, antennas, faster CPUs, and enough memory, can handle around 25 concurrent connected clients, while lesser models choke at at half as many connections. However, since one of our design goals was that it should be as inexpensive as possible to get started with smart home technology, we do not expect a average home Wi-Fi to get saturated right away. The scenario where a homeowner invests in over 20 new thingies at the same time might not be unlikely, but in that case we feel that investment in an extra access point to handle traffic from thingies is probably also within the budget. It is common knowledge among Some Wi-Fi experts claim that Wi-Fi clients also have a tendency to drop their connection from access points more often if using the higher bandwidth 802.11n and 802.11g protocols compared to the slower 802.11b. The solution here is to make thingies use 802.11b where possible, and since only a very small subset of thingies will require a 100% stable connection, homeowners should experience no problems since thingies can simply reconnect if their connection drops.

Communication protocols

For communication, we use standard HTTP [9]. This opens up for a huge list of possible controllers, basically any type of computing device or library that supports HTTP. This is especially important to us since we do not want to be limited to a certain framework, language, or platform, e.g. Java[4] or Android[5]. Other great properties of HTTP is that it is fairly lightweight and stateless protocol [9], which simplifies things considerably on the thingies. HTTP also supports our goal of an open infrastructure; it might very well be the best supported protocol among developers, and this enables the average app developer to build new functionality on top of the infrastructure on their platform of choice. Another benefit is that HTTP communication is very unlikely to be blocked in firewalls compared to other less used protocols, so there is a much bigger chance that things will just work.

However, HTTP's statelessness is also a weakness in common smart home scenarios, where a context change in one thingy should trigger an event in another thingy. With basic HTTP, a thingy (client) waiting for a state change has to continuously poll the other thingy it is monitoring to discover the state change in a timely manner. This is an waste of resources and could potentially deplete the resources of a thingy entirely, especially if more clients are waiting for changes. Clearly, relying only on HTTP alone does not scale. Several solutions to this problem were considered, the biggest contender to be considered being multicast. The advantages of using multicast is that thingies can simply announce state changes on the network and it would not matter if zero, one, or hundreds of clients were interested, the amount of resources required to do so would be the same. Ignoring the problems associated multicast on Wi-Fi, i.e. lost multicast packets are not retransmitted at the MAC layer [5], the reason we did not choose multicast was that the protocol works on a lower level compared to HTTP, and that would limit the number of supported development platforms. We wanted to stay in the Web-stack of technologies so we chose the WebSocket protocol. In the recent years, the WebSocket protocol has been standardized by the Internet Engineering Task Force (IETF) [11] and is in the process of being standardized by the World Wide Web Consortium (W3C) [18]. The WebSocket protocol supports multiplexing bi-directional, full-duplex communications channels over a single TCP connection and is already supported by the latests versions of most major browsers[6], with numerous frameworks and libraries already built and more on the way. The WebSocket Protocol was essentially designed to be a solution for the scenario where a server needs to push information to a client[11] which web developers have been struggling with for many years, so while it might not scale as well as multicast, we feel this is the best overall approach.

[4] `http://java.com`

[5] `http://android.com`

Service Discovery

Most home networks do not have traditional unicast DNS server set up, so we need a different way to discover thingies connect to the network, both during installation and also if and when their IP address changes. We have decided to use mDNS/Bonjour[7] since it offers wide support across platforms and is relatively lightweight and open source. .

Security considerations

The basic trust model for thingies for Dummies is based on the security of the home's Wi-Fi. Assuming the Wi-Fi is probably configured using WPA2 with AES encryption and a reasonable passphrase, there is a clear boundary for the smart home, and an adversary would not be able to pick up data sent to and from thingies without being connected to the network. This also means, that anyone on the network have full control over the thingies attached to it, which can be a problem in some cases. We have discussed the possibility of adding an extra layer of security that would handle such a case. If the concern is only that unauthorised users cannot perform certain actions on a thingy, i.e. only change state, not update configuration, then we need a way to authenticate each user. This could be done using certificates that clients use to authenticate with. The certificate would be generated during the initial configuration of a thingy, and depending on the number of authorisation levels needed for a thingy, e.g. read only, update state, and update configuration, one or more unique certificates would be generated. Then it would be up to the controller devices used during the initial configuration to distribute the generated certificates to authorized users. However, it is not enough only to authenticate users, since the communication between users and thingies are still being transmitted over an open channel which can allow an adversary to perform replay attacks. The solution to this is to secure the communication channel between users and Thinges, i.e. using HTTPS. Adding the extra level of security does raise the hardware requirements on thingies which increases cost and power consumption, so an environment that supports both "open", low risk thingies and "secure" thingies should provide the best of both worlds.

thingies - actuators and sensors

To help us narrow down the responsibilities, requirements, and goals for thingies we looked closely at both the scenario

[6] The WebSocket protocol is supported from Chrome 16, Firefox 11 and Internet Explorer 10.

[7] Bonjour is available on `https://developer.apple.com/opensource/`

described in the introduction of this paper and Edwards et al's [8] work. In summary, thingies should be:

- Easy to install, maintain, and understand
- Independent
- Have a consistent and predictable software interface

The first two points are important because thingies must be installable and manageable by non-technical users. It is also reasonable to assume that a simple thingy with one or two functions is going to be more reliable than a device with several functions. Not allowing thingies to be dependent on other thingies or services simplifies everything, which again helps makes things easier for the non-technical user. Making things simple also means being less dependent on constant firmware updates, which will limit the amount of maintenance required after the initial installation. This is highly important, as Kindberg et al writes, *"You can't reboot the world, let alone rewrite it, to introduce new functionality. Users shouldn't be led into a disposable physical world just because they can't keep up with software upgrades."* [13].

thingies can have running logic inside them, depending on their functionality, e.g. a power plug could be configured to turn on and off at specific times during the day, or turn on a secondary output if a specified amount of power is being drawn from a primary output, but it cannot depend on context outside of its domain, i.e. other thingies. Such logic should exist on controllers.

The last point, have a consistent and predictable software interface, is important since controllers and thingies are loosely coupled to each other. A controller must be able to build the visual interface to a thingy without any prior knowledge of it, no matter if the thingy is just a "read only" sensor, an actuator or a thingy acting as a bridge for other thingies. To do so, a predictable software interface, in our case, an HTTP-based web service interface is required.

How simple can a thingy be and still be useful? That is a question raised in [8] and it turns out, at least from our informal questioning of friends, family, and colleagues, that even a simple thingy that enables remote toggle of a light switch is useful, and that it is better to have many simple things that end-users are able to comprehend individually instead of one big complex thingy.

Requirements

In keeping with the simple theme, we also have very few explicit physical requirements to thingies. They should:

- Have a unique identifier
- Have a passive NFC tag and/or QR code with its ID
- Have a physical reset/install mode button
- Be able to connect to an IP based network

The unique identifier could easily be the MAC address of the Wi-Fi radio or a generic UUID [14]. A unique ID is required since controllers need a way to identify individual thingies independent of how they are named.

The passive NFC tag or QR code and the physical button are all related to the bootstrapping phase, which we will describe later in this section.

Why not require a Wi-Fi radio in thingies? This is covered by the last requirement, but is more generalized, since we want to allow for thingies that connect directly through the LAN interface on the home Wi-Fi router, and also enable the scenario where a ZigBee enabled sensor thingies connect through a specialized ZigBee to Wi-Fi bridge thingy. We do however still require that all thingies can be reached on their own unique URI through the network. This is trivial for thingies connecting directly through Wi-Fi or LAN to the home network – they simply have a standard URI [3] based on their IP address, e.g. `http://192.168.0.42` . In the bridge scenario, the bridged thingies, e.g. ZigBee sensors, may have to share the IP of the bridge. Luckily, the URI schema allows for ports to be defined, so each bridged thingy will be identified by the bridges IP and their own port, e.g. `http://192.168.0.42:1337`, `http://192.168.0.42:1338`, etc.

Web service interface

Each thingy will have a very basic web service interface consisting of a getter and a setter method, and a method to subscribe to be notified of state changes using the WebSockets protocol. Since controllers have no prior knowledge of thingies, thingies also have a dedicated method which returns a structured description of the their abilities. This makes it possible for the controller to render a UI that lets the user see and manage the state of the thingy.

The structured description associates each state variable in the thingy with an internally unique ID. This is what makes it possible to keep the service interface simple, as these IDs are used when querying and setting state. The following list summarizes the web service interface:

- `/get?id=[list of ids]`
  Returns the current internal values in the thingy, possibly filtered by ids.
- `/set?id=<id>&value=<value>`
  Sets the state of a specified internal value in the thingy.
- `/ui`
  Returns the UI description for the thingy.
- `/subscribe=[list of ids]`
  Subscribe to live updates of state changes, possibly filtered by ids.

The data sent to and from thingies are represented as JSON[8]. We chose JSON since it is more lightweight than XML, but still structured. It is also easy to parse with libraries available for practically all platforms and languages, and it is

[8]`http://json.org/`

aligned with our choice to stick with common web technologies when possible.

JSON UI Definition Language (JUIDL)
Inspired by similar work done in the LockIt paper [**?**], we also defined a JSON based UI definition language, tailored to our needs. The goal for our definition language was that it should convey to controllers *what* type of state is available to controllers, but not *how* that state should be represented visually. This is a similar to XML based Web Service Definition Language (WSDL) [6], however, our JUIDL is much more lightweight, especially since we do not support complex objects.

Bootstrapping and installation
Installing new hardware on a home network is far from trivial, especially when that hardware has little or no UI, and few physical buttons to handle configuration. It is a topic being actively researched in the industry, including the Wi-Fi Alliance[9]. Their 2007 attempt, Wi-Fi Protected Setup (WPS)[10] was very promising and would have been our choice for easy bootstrapping, but considering the security problems discovered in 2011 [17], we do not feel comfortable depending on a technology that makes Wi-Fi networks insecure if enabled. Instead, we have devised our own bootstrapping protocol.

Bootstrapping protocol
Thingies have two modes of operations – installation mode and normal mode. A thingy can be put in installation mode by pressing a button, and should also arrive in installation mode from the manufacturer. In installation mode, the Wi-Fi radio of the thingy is put in access point mode, allowing another device to connect directly to it to configure it.

The general bootstrapping process from a user's point of view is very simple. First, the user plugs in and turns on the thingy. Then, the user starts the Home Remote app on his smartphone and, after entering credentials for the Wi-Fi network, clicks "Install new Thingy". The onscreen instructions will prompt the user to scan the QR code or hold the phone close to the thingy (if the phone is NFC enabled). From this point, the phone and the thingy are connected to each other, and the user is asked to give it a name so it is identifiable by humans. The phone also sends the Wi-Fi credentials to the thingy. The diagram in figure 2 illustrates the workflow of the bootstrapping protocol from a technical point of view.

Once the thingy is bootstrapped on to the home network, the thingy can be configured regularly through its configuration screen in the Home Remote app.

As for security during bootstrapping, we do not think there is a likely attack vector. When a thingy is in installation mode and has its Wi-Fi radio in access point mode, it will have a passphrase set. With the passphrase randomized, it is very hard for an attacker to get access to a thingy and bootstrap it before the user is able to. Even if this unlikely scenario should occur, the user still retains full physical control over the device, and is able to turn it off or restart the installation process again.

[9]https://www.wi-fi.org
[10]WPS specifications available for purchase at http://www.wi-fi.org/knowledge-center/articles/wi-fi-protected-setup™

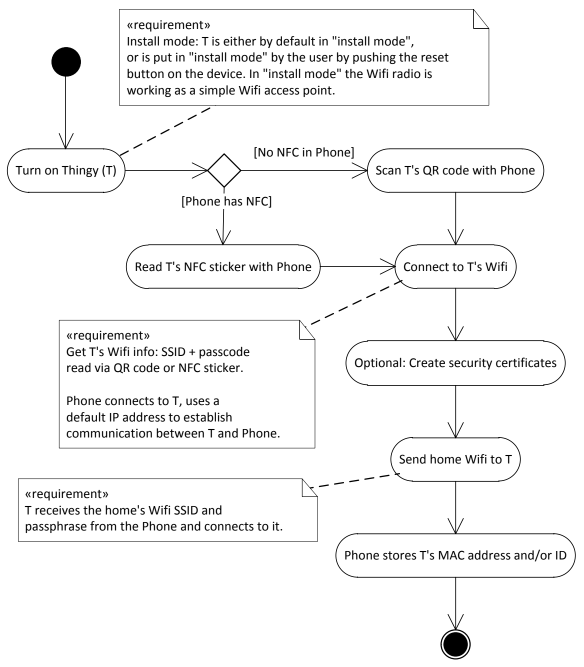

«requirement»
Install mode: T is either by default in "install mode", or is put in "install mode" by the user by pushing the reset button on the device. In "install mode" the Wifi radio is working as a simple Wifi access point.
Turn on Thingy (T)
[No NFC in Phone]
Scan T's QR code with Phone
[Phone has NFC]
Read T's NFC sticker with Phone
Connect to T's Wifi
«requirement»
Get T's Wifi info: SSID + passcode read via QR code or NFC sticker.
Phone connects to T, uses a default IP address to establish communication between T and Phone.
Optional: Create security certificates
Send home Wifi to T
«requirement»
T receives the home's Wifi SSID and passphrase from the Phone and connects to it.
Phone stores T's MAC address and/or ID

Figure 2. Workflow of the bootstrapping protocol.

Controllers
You cannot have a smart home without controllers, and to us, the obvious first choice for controllers are smart phones. It is a device which users carry all the time and are comfortable using. As the previous sections show, we are by no means limited to smart phones – any computing device supporting the HTTP protocol can act as a controller. This is also our only requirements for simple controllers, although we do require a full controller to be able to perform bootstrapping of new thingies, which is where smartphones really shine. To enable the simplest form of user programming and automation on the controller level, and to make the scenario described in the introduction possible, we need a way to execute a batch of commands to different thingies. Our solution to this is something we have dubbed "Presets". A Preset consists of a name and a set of thingies, along with a desired state for each thingy. When the user defines a new Preset, he gives it a name and then adds one thingy at a time to the preset, specifying which state it should be in.When a Preset is triggered by the user, all thingies in the preset are put in the required state in sequential order, and the expected state of each thingies is checked. If any thingies in the Preset did not end up in the expected state, the user is notified. This also applies to thingies that are only sensors, where the Preset is simply used to verify their states. This enables relatively advanced scenarios such as our "Lock Down" scenario described earlier.

This is clearly just the tip of the iceberg in terms of automation. It is easy to imaging additional functionality and in some cases it is simply a matter of enhancing the Preset concept, e.g. having Presets triggered at certain times. When designing more complex automation, there are other issues to consider; e.g. active monitoring of sensors might drain the battery of the phone too fast.

RELATED WORK

The Internet of Things, implemented in e.g. HP's Cooltown project[12] is a concept related to ours in that it connects the physical and virtual worlds by creating an online presence for physical objects. The Cooltown project also mentions the use of local wireless networks as a means of access restriction.

Smart home technologies

X10[11] is a low-level protocol from the 70s running through the power lines in a home. It allows one to control a number of devices using a pre-defined set of commands [12]. Thus, it requires a controller device which is connected to the power lines. The power line medium also introduces potential issues with X10 commands leaving or entering a home.

The Open Services Gateway Initiative, OSGi[15], is an infrastructure designed by an alliance of over 80 companies. [7] describes how the technology can be used in the context of a smart home, including integration with devices through other technologies such as UPnP and Jini. The approach makes use of a gateway device within the home network. The infrastructure is marketed towards vendors and used in various products and solutions[2].

Table 1 compares some properties of our approach versus the ones we have mentioned:

	Installation by end-users	Security	Wireless control	Complexity
X10	Yes, with limitations	Relies on signals not leaving or entering the home	Requires extra equipment	Low
OSGi	Yes, for some products	Varies with products	Varies with products	High
Thingies for Dummies	Yes	Provided by Wi-Fi network	Yes	Low

Table 1. Comparison of smart home approaches

Sources for X10: [13], [14]. Sources for OSGi: [2], [7].

[11]`http://x10.com`

[12]urlhttp://software.x10.com/pub/manuals/xtdcode.pdf

[13]`http://www.smarthome.com/about_x10.html`

[14]`http://kbase.x10.com/wiki/MyHouse_Online_Remote_Connection_Setup`

OSGi is a complex beast with many existing solutions. We have not been able to find one that matches ours in terms of simplicity. When it comes to simplicity and end-user installation, X10 is the approach most similar to ours. However, there is the caveat that X10 signals can enter and leave a home through the power lines, so for a secure installation, an electrician is required to install a filtering device. Additionally, X10 does not have the convenience of wireless control built in; this requires the addition of a controller device that supports this.

User interface generation

[13] defines four levels of intelligence when moving user interfaces across devices, ranging from e.g. VNC[16] where raw pixels and input events are transferred, to e.g. Jini[19] where the Java code that generates the UI is transferred. In-between, there are approaches where individual widgets are transferred, either by specifying precise widgets (i.e. "this is a drop down"), or on a purely semantic level (i.e. "this control must allow selection among these values"). We position ourselves in line with the latter, semantic approach.

[4] describes an approach for defining defining mobile user interfaces for web services, by adding two layers on top of the web service definition: one containing a semantic description of the service, and one containing a semantic description of the UI. While this approach may be more generally applicable than ours, it also introduces a considerable complexities.

IMPLEMENTATION

We have implemented a subset of our designed infrastructure. The implementation includes a sample thingy in the form of a light switch, and a controller in the form of an Android app. We have implemented bootstrapping of the thingy onto one's home Wi-Fi network as well as communication between the controller and the thingy.

We have not implemented secure thingies with user authentication, or synchronization of configuration across users; nor is the Android-side of the UI generation implemented beyond what is needed for the sample thingy. The sample thingy requires no configuration, so this has been left out. Also, remote access from outside the Wi-Fi network is not implemented, and we opted to use HTTP polling rather than web sockets for communication with the thingy. Finally, the bootstrapping process was implemented differently; more on that later.

Our implementation is described in the following.

Controller: Home Remote for Android

We implemented a controller Android app, dubbed Home Remote. Its initial view (see figure3) shows a list of installed thingies along with an option to install new ones. Selecting a thingy shows the user interface for that thingy (same figure).

Presets

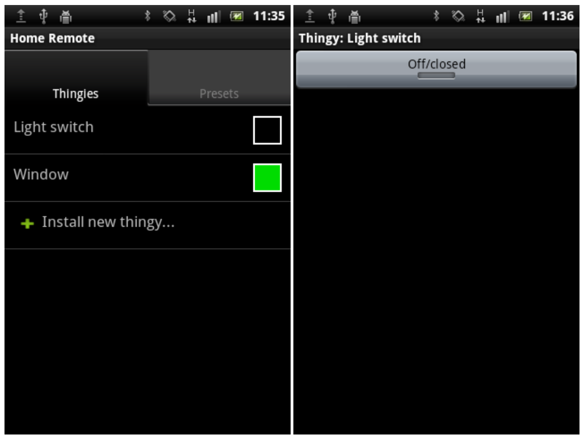
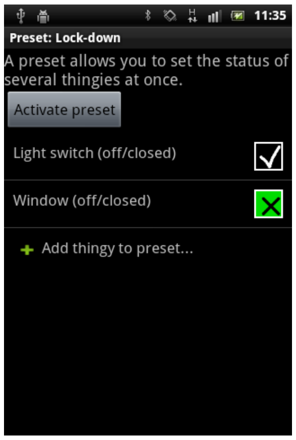


Figure 3. Left: The initial view in our controller app. The status of each thingy is shown in a square. Right: The interface for one thingy.

Presets have been implemented as designed – a preset consists of a list of thingies along with a desired status for each thingy (figure 4). Activating a preset causes thingies with actuators to change state, while thingies with sensors are listed to let the preset function as a check list. Check marks are overlaid on the status icons of thingies to indicate whether each thingy is in the desired state.



A preset allows you to set the status of several thingies at once.

Figure 4. A preset in our controller app. The light is off as desired, but the window is still open.

Thingy: Light switch

Our sample thingy, a light switch, was implemented on an Arduino board, with a WiFly shield[15] for Wi-Fi connectivity. A relay was used in a circuit based on this one[16], except with a 480Ω resistor and two transistors mounted as a Darlington pair[17]. Once installed, this thingy allows one to control a power outlet and thus, for instance, a lamp.

The bootstrap process

Recall that the purpose of our bootstrap process is to get a new thingy connected to the home Wi-Fi. In the design, we had envisioned a bootstrap process where the thingy would provide a temporary access point that the Android app could connect to for initial communication. However, it turned out that while the WiFly board supports ad hoc mode, it cannot work as a regular access point[18]. Ad hoc mode is, in turn, not supported by Android phones out of the box[19]. However, since Android phones *are* able to work as regular access points, we decided to turn the process around:

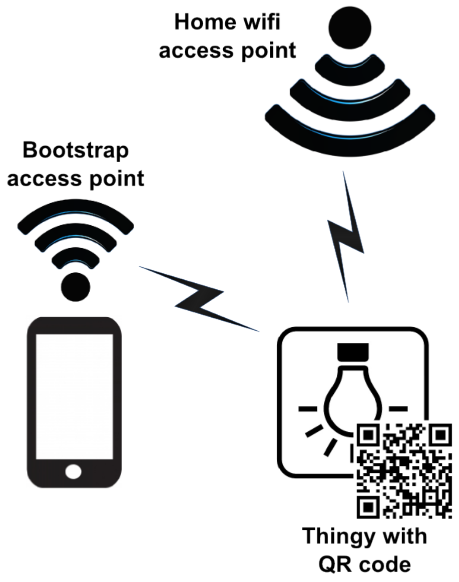


Figure 5. Devices and access points involved in the bootstrap process

When the user starts the process by selecting “Install new thingy” in the Android app, he is first prompted to scan the QR code that came with the thingy. This code contains the SSID and passphrase of the bootstrap access point that the app then sets up on the phone. Second, the app instructs the user to press the button on the thingy; this causes the thingy to make a request to the Android phone via the bootstrap access point. The response to this request contains the SSID and passphrase of the home Wi-Fi access point, which the thingy then connects to, obtaining an IP address. Next, the thingy briefly reconnects to the bootstrap access point to send this IP address to the Android app. Finally, the thingy reconnects to the home Wi-Fi, and the Android app closes the bootstrap access point and reconnects to the home Wi-Fi as well. Thus, the thingy is connected to the home Wi-Fi and the Android app is able to make requests to the thingy.

One disadvantage of this implementation is the use of an undocumented API call[20], another is the fact that older Android phones cannot work as access points, and third it relies on the home access point issuing the same IP address when

[15]`http://www.sparkfun.com/products/9954`

[16]`http://www.arduino.cc/playground/uploads/Learning/relays.pdf`

[17]`http://en.wikipedia.org/wiki/Darlington_transistor`

[18]`http://www.sparkfun.com/datasheets/Wireless/WiFi/WiFlyGSX-um2.pdf`

[19]`http://szym.net/2010/12/adhoc-wifi-in-android/`

[20]`http://stackoverflow.com/questions/3023226/android-2-2-wifi-hotspot-api`

the thingy reconnects. That being said, it works, and the user experience is the same as we had envisioned.

EVALUATION

The evaluation in our prototype was performed at three points: first, an initial evaluation of the overall idea, second, an evaluation of an early prototype using the "Wizard of Oz" technique, and third, an evaluation of the final prototype. Each is described in the following.

Initial evaluation of the idea

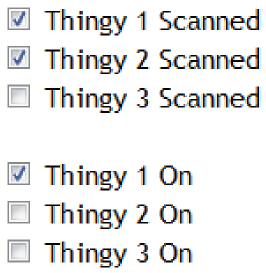
Before setting out to develop a prototype, we wanted to be sure that our idea was useful. Therefore, we checked with a number of friends and colleagues. We used short, semi-structured interviews, asking "Would you be interested in controlling something in your home from your phone, and if so, what would it be?". We also told them that controlling objects while away from home was a possibility. Five participants were asked.

We generally received a moderate amount of interest – it was not the case that everyone was instantly thrilled by the possibility, but all participants did think of something that they would actually like to remote control, either from inside or outside the home. The ideas are listed in the introduction.

This assured us that it was worthwhile to proceed.

Evaluation of the first prototype

In order to evaluate a prototype as early as possible, we decided to use a "Wizard of Oz" – approach where we faked the parts of the system that were the least perceptible for the participants. We thus implemented the interface for the Home Remote Android app that the participants would interact with directly. A fake thingy was built in the form of a light switch – in reality, a physical switch at the end of an extension cord allowed the test conductor to turn the light on or off according to the participant's interaction with the app. Similarly, the association of the fake thingy with the home Wi-Fi was simulated. A simple web interface, shown in figure 6 let the test conductor see and change the status of thingies that was shown in the app.


- ✓ Thingy 1 Scanned
- ✓ Thingy 2 Scanned
- ☐ Thingy 3 Scanned

- ✓ Thingy 1 On
- ☐ Thingy 2 On
- ☐ Thingy 3 On

Figure 6. Web interface for the wizard. The "scanned" boxes indicate whether the thingies are connected to Wi-Fi while the "on" boxes indicate their status.

The evaluation focused on the installation process, the concept of remote control, the concept of presets, and the overall usefulness of our idea. An installation guide[21] was handed out. In order to evaluate the ease of installation, we selected two less technical, middle-aged participants, one of which already owned a smart phone. While the number of participants for this evaluation was indeed limited due to time constraints, we still believe that the results are useful – especially where there are similarities in the reactions. For details on how this evaluation was performed, see the evaluation plan. It is available online[22].

The "Wizard of Oz"-approach worked surprisingly well as participants forgot about the extension cord and were surprised to see the the light control working. This system introduced a new causal relationship[10], i.e. interacting with the app caused lights to go on and off, and participants quickly picked up on this.

Neither participant found it immediately obvious how to control a thingy from the app after having installed the thingy, so the interface needs some tweaks in this regard. The concept of presets required some explaining, and with only one physical thingy it proved difficult to provide the illusion of controlling several thingies at once.

Both participants were mostly interested in remote scenarios – e.g. checking that everything was in order after having left the home. This shows us that we do indeed need to cross the boundary of the home Wi-Fi, perhaps through a cloud service or a gateway device in the home.

Evaluation of the final prototype

For the final prototype, we set out to implement a light switch. With the bootstrapping implemented, we were able to to evaluate the installation process. As we were, despite an enthusiastic attempt, unable to build the circuitry to control an actual switch in time for the evaluation, we opted not to evaluate regular use at this point.

We selected two participants in their early twenties. Both had experience with smart phones but were not overwhelmingly tech savvy. Neither participant had been part of the previous evaluation. We used the same questions as previously, skipping the ones that did not apply.

Both participants found that the process was simple and easy to follow overall, one stating that the process was well-guided by the app. One participant pointed out that there was a lack of feedback during the process – the bootstrapping took around 20 seconds with a static text indicating that it would take "a little while". This was especially apparent when the thingy failed connecting in the first attempt and required an extra button push. The other participant had not previously scanned a QR code and thus found this a little less intuitive than we had anticipated.

DISCUSSION

[21]`https://github.com/egil/DEX1/blob/master/report/extras/installation-guide.pdf`
[22]`https://github.com/egil/DEX1/blob/master/report/extras/evaluation-plan.pdf`

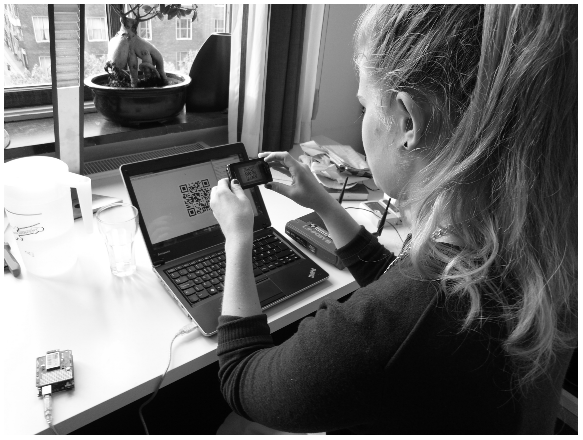


Figure 7. Evaluation of the bootstrapping process.

OUR DREAMS NOTES: It is on the controller that we envision the platform to take shape in a similar way to e.g. the Android platform. We imagine that the technologies that developers already understand will enable new innovative ways to control the home. [More about cloud here.]

- Comparing home smart thingies to office thingies, i.e. usage patterns.

FUTURE WORK

Cloud backed solution - adds real remote control support - would require a local "bridge" between inhouse thingies and cloud provider

CONCLUSION

TODO

ACKNOWLEDGEMENTS

We would like to thank Jakob Bardram, Sebastian Büttrich, and especially Thomas Pederson for excellent supervision. A thank-you is also due for the guys at Brinck's Elektronik[23], and to those who participated in our evaluation.

REFERENCES

1. Z. Aliance. ZigBee zigbee aliance. `http://www.zigbee.org/`. [Online; accessed 1-May-2012].
2. O. Alliance. OSGi alliance – markets: Smarthome. `http://www.osgi.org/Markets/SmartHome`. [Online; accessed 6-May-2012].
3. T. Berners-Lee, R. Fielding, and L. Masinter. Rfc 3986, uniform resource identifier (uri): Generic syntax. Request For Comments (RFC), 2005.
4. G. Broll, S. Siorpaes, E. Rukzio, M. Paolucci, J. Hamard, M. Wagner, and A. Schmidt. Supporting mobile service usage through physical mobile interaction. In *Pervasive Computing and Communications, 2007. PerCom '07. Fifth Annual IEEE International Conference on*, pages 262 –271, march 2007.
5. R. Chandra, S. Karanth, T. Moscibroda, V. Navda, J. Padhye, R. Ramjee, and L. Ravindranath. Dircast: A practical and efficient wi-fi multicast system. In H. Schulzrinne, K. K. Ramakrishnan, T. G. Griffin, and S. V. Krishnamurthy, editors, *ICNP*, pages 161–170. IEEE Computer Society, 2009.
6. E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web service definition language (wsdl). Technical Report NOTE-wsdl-20010315, World Wide Web Consortium, March 2001.
7. P. Dobrev, D. Famolari, C. Kurzke, and B. Miller. Device and service discovery in home networks with OSGi. *Communications Magazine, IEEE*, 40(8):86 – 92, aug 2002.
8. W. K. Edwards and R. E. Grinter. At home with ubiquitous computing: Seven challenges. pages 256–272. Springer-Verlag, 2001.
9. R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – http/1.1, 1999.
10. Y. Hagmayer, S. A. Sloman, D. A. Lagnado, and M. R. Waldmann. Causal reasoning through intervention. In A. G. . L. Schulz, editor, *Causal learning: Psychology, philosophy and computation*. Oxford University Press, 2007.
11. I. E. T. F. (IETF). IETF the websocket protocol. `http://tools.ietf.org/html/rfc6455`. [Online; accessed 4-May-2012].
12. T. Kindberg, J. Barton, J. Morgan, G. Becker, D. Caswell, P. Debaty, G. Gopal, M. Frid, V. Krishnan, H. Morris, J. Schettino, B. Serra, and M. Spasojevic. People, places, things: Web presence for the real world. *Mobile Networks and Applications*, 7:365–376, 2002. 10.1023/A:1016591616731.
13. T. Kindberg and A. Fox. System software for ubiquitous computing. *Pervasive Computing, IEEE*, 1(1):70 – 81, jan-mar 2002.
14. P. Leach, M. Mealling, and R. Salz. Rfc 4122: A universally unique identifier (uuid) urn namespace, 2005.
15. D. Marples and P. Kriens. The open services gateway initiative: an introductory overview. *Communications Magazine, IEEE*, 39(12):110 –114, dec 2001.
16. T. Richardson, Q. Stafford-Fraser, K. Wood, and A. Hopper. Virtual network computing. *Internet Computing, IEEE*, 2(1):33 –38, jan/feb 1998.
17. S. Viehbck. Brute forcing wi-fi protected setup. when poor design meets poor implementation. `http://sviehb.files.wordpress.com/2011/12/viehboeck_wps.pdf`, dec 2011. [Online; accessed 1-May-2012].

[23] `http://elextra.dk`

18. W. W. W. C. (W3C). W3C the websocket api. `http://www.w3.org/TR/websockets/`. [Online; accessed 4-May-2012].

19. J. Waldo. The jini architecture for network-centric computing. *Commun. ACM*, 42(7):76–82, July 1999.

20. M. Weiser. The computer for the 21st century. *Scientific American*, 3(3):3–11, 1991.